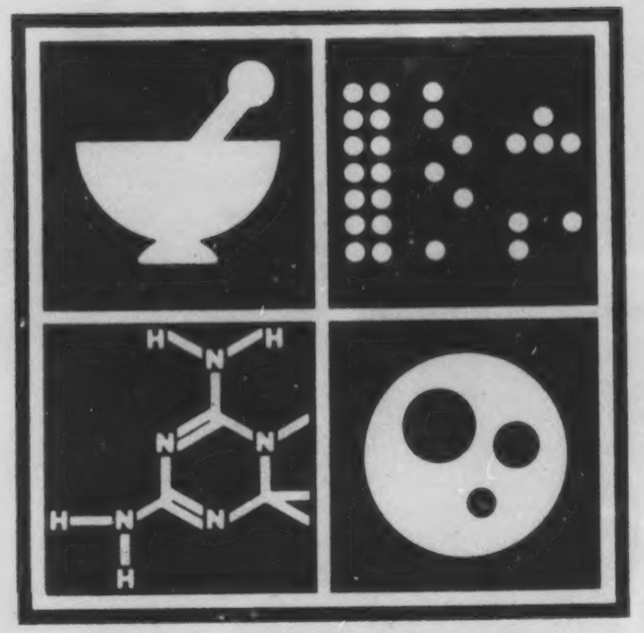


PROPHET

P/L PROPHET

U.S. DEPARTMENT OF HEALTH AND HUMAN SERVICES

Public Health Service
National Institutes of Health

P/94/6

PL/PROPHET

A User's Guide to PL/PROPHET:
the PROPHET System's Programming Language

DIVISION OF RESEARCH RESOURCES

NATIONAL INSTITUTES OF HEALTH

Prepared by Bolt Beranek and Newman Inc., Cambridge, Mass.
Under Contract # N01-RR-5-2101
Biomedical Research Technology Program
Division of Research Resources
National Institutes of Health

REVISED SEPTEMBER 1985

NEWARK PUBLIC LIBRARY

MAR 14 1986

U. S. GOVT. PUBS.
DEPOSITORY 376

HE20.3008:

TABLE OF CONTENTS

Page

Introduction i

I. PL/PROPHET Basics 1-1

- A. Characters 1-1
- B. Simple Data Types 1-3
- C. Variables in PL/PROPHET 1-9
- D. PL/PROPHET Operators 1-10
- E. Arrays 1-15
- F. Complex Data Types 1-18

II. Basic PL/PROPHET Statements 2-1

- A. Assignment Statements 2-1
- B. Input/Output Statements 2-3
- C. Conditional Statements 2-7
  - 1. The IF . . . THEN Statement 2-7
  - 2. The ELSE Statement 2-8
- D. The DO . . . END Block 2-11
- E. The DO Loop 2-13
- F. The GO TO Statement and LABELs 2-23
- G. The RETURN Statement 2-27
- H. The CLEAR Statement 2-28

III. Basic PL/PROPHET Functions and Procedures 3-1

- A. The EMPTY and NOTEMPTY Functions 3-1
- B. Data Conversion Functions 3-3
- C. The COPY Function 3-6
- D. Arithmetic Functions 3-6
- E. Text Functions 3-10
- F. Type-Checking Functions 3-12
- G. Input/Output Functions 3-13
- H. Array Functions 3-16
- I. Date and Time Functions 3-18
- J. System Variables 3-19

READEX FICHE best copy available

Page

IV. Procedures in PL/PROPHET 4-1

A. The Definition of a Procedure 4-1

1. Comments in a Procedure 4-1
2. Opening and Closing Lines 4-3
3. Procedure Names 4-4
4. Arguments of a Procedure 4-5
5. Declarations in Procedures 4-6
6. Functions 4-9
7. Multiple Entries in Procedures 4-11
8. Internal Procedures 4-13

B. Notes of a Procedure 4-16
C. Compiling Procedures 4-16
D. Using Procedures 4-18

V. Writing and Debugging PL/PROPHET Procedures . . . 5-1

A. Programming Guidelines 5-1
B. Common Compile-Time Problems 5-7
C. Common Run-Time Problems 5-11
D. Efficiency Considerations 5-15
E. Debugging Techniques 5-22

VI. Advanced PL/PROPHET Statements 6-1

A. The NOEVAL Statement 6-1
B. The EVAL Function 6-1
C. The BEGIN . . . END Block 6-3
D. The %ERRORFLAG Statement 6-3
E. The ERROR START Statement 6-5

VII. Table Functions and Procedures 7-1

A. Moving Data To or From a Table Cell 7-2
B. Operations on the Structure of a Table . . . 7-3
C. Operations on the Structure of a Column . . . 7-7
D. Making Tables Within a Procedure 7-12
E. Adding Columns to Tables 7-15
F. Allocating Space for Tables 7-17
G. Deleting Columns, Rows, and Tables 7-18
H. Table/Array Data Transfer 7-19
I. Miscellaneous Table Functions and Procedures 7-23

Page

VIII. Display Procedures and Functions 8-1

IX. Molecule Functions and Procedures

(See Molecules Manual)

X. Statistical Functions and Procedures

(See Statistics Manual)

XI. Communicating with Other Computer Programs . . . 11-1

- A. Communicating with Other Programs from PROPHET 11-2
- B. Communicating with Other Computers. 11-4
- C. Using Disk Files 11-6
- D. Using Formatted Data 11-12
- E. An Example of Extra-PROPHET Communication and File Handling . 11-20

XII. Vector, Matrix and Transformation Procedures and Functions 12-1

- A. Representation of Vectors and Matrices . . 12-2
- B. Algebraic Operators 12-4
- C. Vector and Matrix Functions 12-8
- D. Transformers 12-11
- E. Miscellaneous Vector and Matrix Procedures. 12-16
- F. Example of Vectors and Matrices 12-19
- G. Implementation of Transformers 12-24

Page
XIII. Miscellaneous Procedures and Functions 13-1

A. Input/Output Procedures and Functions . . . 13-1
B. Sorting Procedures 13-3
C. Text Functions 13-5
D. Array Procedures 13-6
E. Date and Time Functions 13-7
F. Terminal Functions and Procedures 13-10
G. Batch Aids 13-11

Index

Introduction

The User's Guide to PL/PROPHET is intended to serve as documentation of the PROPHET System's programming language. An increasing number of PROPHET users are writing procedures to solve their problems; many of them have expressed a need for more comprehensive documentation of the many programming tools available in PL/PROPHET than is possible in the framework of the User's Manual.

If your interest in procedures is limited to restoring public procedures and calling them, you will probably not need to use this manual--the Public Procedures Notebook will usually serve your needs most adequately. If you are interested in writing your own programs, however, we hope that this new document will assist you in making efficient use of the powerful features available in PL/PROPHET.

Acknowledgements

I would like to acknowledge the assistance and encouragement provided by PROPHET users in the development of this manual. I also want to thank the PROPHET team at BBN for their helpful suggestions and advice, particularly Dr. Charlotte Hollister, whose editorial supervision was invaluable. Last, but by no means least, I want to thank Ms. Elizabeth Raymondi, who skillfully prepared the text of this document on the computer.

I PL/PROPHET Basics

A. Characters

PL/PROPHET recognizes the following characters:

1. all upper-case alphabetic characters from A to Z;
2. the numeric characters 0 through 9;
3. the special characters

   , . / [! " # $ % & ' ( ) : * - =] < > ? @ ^ \ `

   (Some of these characters have special meanings defined below.)
4. The formatting characters Space (produced by hitting the spacing bar), TAB, CARRIAGE RETURN, and LINE FEED. (The character produced by hitting the Return key is actually two characters -- a carriage return and a line feed.)
5. Control characters (discussed below) which are produced by striking a key while holding down the CTRL key.

Special Characters

Special characters are characters which have, in addition to their text value, some special function within PL/PROPHET. The actual use of the various operators is discussed in Section I.D. These characters and their special functions are:

<GO>	Terminating characters
;	
!	
	TEXT delimiter
+	Arithmetic operator
-	Arithmetic operator
*	Arithmetic operator
/	Arithmetic operator
=	Relational or assignment operator
<	Relational operator
>	Relational operator
.	Decimal point and Text operator
()	Used to group expressions and arguments and to indicate array subscripts
[]	Used to group array constants
:	Used in Procedure Definitions to indicate Procedure Names, Entries, and Labels

Control Characters

There are also special Control Characters which are obtained by holding down the key on the terminal marked CTRL while striking an alphabetic key. These control characters have special functions, which are described briefly below, and in greater detail in the PROPHET Users Manual.

Control-C	Monitor Key
Control-G	Bell
Control-I	Tab
Control-L	Form Feed or Page
Control-O	Output Interrupt
Control-Q	Stop or Abort Key
Control-R	Text Input Key
Control-S	Empty Key
Control-T	Status Key
Control-U	Delete Key (deletes entire line)
Control-Z	End-of-File

All these Control Characters carry out functions which are both useful in PROPHET and not obtainable in other ways. Note however that any Control character probably has a special function. Therefore do not use random Control-characters.

Other Characters

Depending on the kind of terminal you are using, there may be some other keys with special functions: e.g., a RUBOUT key, which deletes a single character. Other keys such as LINE FEED, BREAK and RESET are not generally used in PROPHET.

B. Simple Data Types

Within PL/PROPHET, each piece of data is categorized as being of some specific data type. The data type implies a specific structure internal to the computer, pre-formatted input and output patterns, and operations defined especially for the data type. The basic data types in PL/PROPHET are:

FIXED
FLOAT
TEXT
BOOL
ANY

The first four data types and their properties are summarized in Table 1.1. The data type ANY is discussed below.

Table 1.1 SIMPLE DATA TYPES

Data Type	Description	Constants
FIXED	Fixed point integer	Constants of type FIXED consist of a sequence of digits with an optional + or - sign. A decimal point is not allowed. The largest FIXED the system can handle is 35359738367; the smallest FIXED the system can handle is -35359738368 (if you wish to have a larger or smaller number, it must be a FLOAT). Examples: 12 4096 0 but not 87667876543 (too big)
FLOAT	Decimal number; floating point	Constants of type FLOAT have two basic forms: 1) The "fixed-point" form is any string of digits containing a decimal point somewhere in the string. For example 12. .0345 234000.123 but not 422 (no decimal point). 2) The "floating-point" form consists of a constant of either type FIXED or type 1) FLOAT (above) followed by the letter E followed by an optional sign followed by a constant of type FIXED. The FIXED after the E indicates a power of 10 by which the number before the E must be multiplied to yield the desired number. For example 1E12 is the same as 1000000000000. 1.56E-4 is the same as .000156 1.342177E6 is the same as 1342177. The range of representable FLOATs is 1.47*10**-39<(absolute value) <1.70*10**38, plus 0. The mantissa of a FLOAT is stored in 27 bits and therefore may contain as many as 8 (or sometimes 9) significant digits.

Table 1.1 SIMPLE DATA TYPES (cont.)

Data Type	Output Format
FIXED	Output as a decimal integer, with leading zeroes suppressed, no leading or trailing blanks, and no decimal point. A sign is printed only if negative. If the value is 0, then one zero is output.
FLOAT	1) If the value is an integer, it is output exactly as with FIXED, except that a decimal point is included. 2) If 10**-4<=(absolute value)<10**6 then the integral part is printed (with a sign printed only if negative), followed by a decimal point, followed by the fractional part with trailing zeroes suppressed. The number is truncated after 8 or 9 significant digits. 3) If (absolute value) >10**6 or <10**-4 then the value is printed as a FLOAT of form 1) or 2) above (normalized to be between 1.0 and 10.0) followed by an E followed by an integer representing the power of 10. For example, 123456.71 would be printed as 1.234561E5

Table 1.1 SIMPLE DATA TYPES (cont.)

<table>
<tr><th>Data Type</th><th>Description</th><th>Constants</th></tr>
<tr><td>TEXT</td><td>A string of ASCII characters</td><td>A constant of type TEXT consists of the desired sequence of characters enclosed between two single quotation marks (the character '). Every character between the quotation marks is significant, even spaces and carriage returns. If an apostrophe (single quotation mark) is to be in a TEXT, it must be written as two single quotation marks with no intervening blank (not a double quotation mark). Examples:
<pre>
'A* CDE'
'THE QUICK RED FOX JUMPED OVER THE
            LAZY BROWN
     DOG'
''   (the "null" text)
'BEETHOVEN''S "5TH SYMPHONY"'
     (this last example would type out
     as BEETHOVEN'S "5TH SYMPHONY")
</pre></td></tr>
<tr><td>BOOL</td><td>Boolean value</td><td>There are just two BOOL values: TRUE and FALSE. For example, if FLAG is a BOOL, then
<pre>
          FLAG = TRUE;
</pre>
will, upon execution, set FLAG to the boolean value TRUE.</td></tr>
</table>

Table 1.1 SIMPLE DATA TYPES (cont.)

TEXT	The value is printed. For example the TEXT 'ABCDE' is printed as ABCDE
BOOL	The value is printed as either TRUE or FALSE

The Data Type ANY

In PL/PROPHET, a variable or constant must be of only one data type. In some cases this restriction is inconvenient. For example, you might want to allow a parameter in a procedure to be of any of the data types in Table 1.1, and to take different actions, depending on the type of the variable at execution time. If the logic of a procedure is substantially the same for a number of data types, then it would be convenient not to have to repeat the program in several slightly different versions. You can declare the parameter to be of type ANY and use the function TYPETEST (described below in Section III) within the procedure to determine which action to take. Note, once the variable is given a value, it has the corresponding data type until it is set to another value. Thus, ANY is not a data type in the true sense, but a variable of type ANY can take on the appropriate data type when required.

Data Type Conversions

There may be times when you will want to have a piece of data of one type available as a different data type. PL/PRCPhET will implicitly convert data to the appropriate type in many cases, but it is usually better to explicitly convert data from one type to another by using the appropriate conversion functions described below in Section III.B.

C. Variables in PL/PROPHET

Data in PL/PROPHET fall into two general categories: constants and variables. A constant is a piece of data whose value is given explicitly and which cannot change during the course of a program. A variable is a piece of data with a name, and may take on different values during the execution of a program, though it need not do so.

If a piece of data is a constant, its data type can usually be inferred from its value. If the piece of data is a variable, its data type must usually be specified in a declaration statement (see Section IV).

Naming Variables

A variable is referred to by a variable name. This name may consist of an arbitrarily long string of alphanumeric characters (including the characters "%" and "$"). It cannot include a space or any of the special characters listed in Section I.A. The first character of a variable name must be alphabetic. "%" and "$" are valid as the first character of a variable name, but you should avoid using "%" since it is used as the first character of many PROPHET system functions and procedures. Note that unlike FORTRAN, for example, variables whose names begin with the letters I through N are not automatically assumed to be of typed FIXED.

Reserved Words

There are also some words reserved by PROPHET for its own use which you cannot use as variable names. These are listed in Appendix I of the Users Manual.

D. PL/PROPHET Operators

Within PL/PROPHET, there are several operators which permit variables to be combined into expressions, or perform operations on combinations of variables.

1. Arithmetic Operators

There are six arithmetic operators:

+	used for addition (as an infix), or for designation of sign (as a prefix),
-	for subtraction (as infix), for negation (as prefix),
*	for multiplication,
/	for division
//	for integer division
**	for exponentiation

All of the arithmetic operators may be used to form FIXEDs and FLOATs into expressions. As in standard mathematical notation, parentheses may be used to indicate explicit groupings.

Precedence of Arithmetic Operators

An arithmetic expression is evaluated in the order of precedence of the operators. The *prefix* + and - operations have the highest (and equal) precedence and will be completed first. Next comes the ** operation, followed by *, /, and // (which have equal precedence); followed by the *infix* + and - (of equal precedence). If there is more than one **, they are evaluated from *right* to *left*. If there is more than one operator at one of the other precedence levels, they are evaluated from *left* to *right*. Parentheses can be used to change the order of evaluation. The innermost parenthesized material is evaluated first. For example,

A + B + C   is equivalent to   (A + B) + C

A + B * C   is *not* equivalent to   (A + B) * C

Whenever a complicated expression is to be evaluated, parentheses should be used freely to insure that the actual expression evaluated is the one intended.

FIXEDs and FLOATs (including PL/PROPHET functions which return a FIXED or FLOAT value) can be mixed in arithmetic expressions, but it is always more efficient to create an arithmetic expression of all FIXEDs or all FLOATs (except for exponentiation where a FIXED power is usually intended). For example,

```
3.2**4
-.0345
4**3.2
A + SIN(1.0) (where A is either FLOAT or FIXED)
```

are all valid arithmetic expressions.

The results of a calculation involving FIXEDs and the / operator will be a FLOAT. All other arithmetic operators, when operating exclusively on FIXEDs will produce a FIXED result. For example, the expression 3/2 has the value 1.5 (a FLOAT). To obtain a truncated integer result, the // operator must be used. 3//2 has the value 1 (a FIXED).

2. Text Operators

TEXT variables and constants may be formed into expressions by just one operator: concatenation. The concatenation operator juxtaposes the characters of one TEXT with the characters of another TEXT. For example if the TEXT variable A has the value 'GRAPE' and the TEXT variable B has the value 'FRUIT', then the expression

```
A CAT B
```

has the value 'GRAPEFRUIT'. An equivalent form of the concatenation operator is

A.B

that is, the concatenation operator may be represented by either "." or by "CAT".

3. Relational Operators

The Relational Operators are used to test the relationship between two variables. Unlike most operators Relational Operators are not restricted to variables of a specific data type. The Relational Operators are:

=	equal
<>	not equal (>< can also be used)
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

= and <> may be used with data of any type, and the data they connect need not be of the same data type. The relational expression

```
expression1 = expression2
```

is TRUE if "expression1" and "expression2" hold data of the same type and of equal value.

A <> B is identical in value to NOT(A = B).

If A and B are expressions of different data types, then the expression A = B is always FALSE, and the expression A <> B is always TRUE.

The operators >, <, <=, and >= may be used with data types that have a linear ordering; viz., TEXT (using the alphabetical order given in Appendix III of the Users Manual), FLOAT, and FIXED data, but only one data type may appear in a given expression.

The item on either side of the operator may be a constant, a variable name, or an expression. In

Section II.A, the use of the relational operators to construct relational expressions is given close examination.

4. Boolean Operators

Boolean expressions are composed of boolean data (including relational expressions and functions which return a BOOL) connected by the operators

AND
OR
NOT
XOR

The truth tables for AND, OR, and XOR are:

OR	T	F
T	T	T
F	T	F

AND	T	F
T	T	F
F	F	F

XOR	T	F
T	F	T
F	T	F

The operator is shown in the upper left-hand corner, the values of the two inputs along the top and the left-hand side, and the result in the middle of the table. Thus if X is equal to 2, then the expression ((X > 5) OR (X <3)) has the value TRUE, while ((X > 5) AND (X < 3)) is FALSE.

Precedence of Boolean Operators

A given relational expression may also be preceded by the NOT operator. NOT is a prefix operator which turns the value TRUE into FALSE, and the value FALSE into TRUE. For example, if X is equal to 2, then NOT(X > 5) has the value TRUE and NOT(X < 3) is FALSE. As with arithmetic expressions, logical expressions are evaluated in a fixed order. All expressions involving NOT are evaluated first; then all expressions involving AND. Expressions involving OR and XOR are evaluated last. Within groups of the same precedence, evaluation takes place from left to right. Parentheses may be used to change the order of evaluation since any expression within parentheses is evaluated first, following the rules given above. Thus, using X = 2 again, NOT (X < 3) OR (X > 5) equals FALSE, but NOT((X < 3) AND (X > 5)) equals TRUE.

E. Arrays

An ARRAY is an organized set of objects, any one of which can be accessed by its subscript or index within the set. Thus if ARA is defined as a FLOAT ARRAY, it is said to be a set of FLOAT numbers, any one of which can be addressed by its subscript. Values can be assigned into an array by means of a subscript; e.g.,

```
ARA(3) = 3.14
```

and values can also be assigned from an array using the subscript:

```
X = ARA(4)
```

Arrays can be organized from <u>any</u> data type in PROPHET.

ANY Arrays

If you want an array to hold more than one data type--TEXT and FLOAT, for example--you must declare the array to be of type ANY.

Multi-Dimensional Arrays

When an array is defined as an array of arrays, it is said to be a multi-dimensional array. Thus, if A is a FLOAT array containing the numbers [1.,2.,3.,4.] and B is a FLOAT array containing the set of numbers [8.,9.,10.]; we can construct a FLOAT array array (or a two-dimensional FLOAT array) called MTR by using the statements:

```
DECLARE MTR(*,*) FLOAT
MTR(1) = A
MTR(2) = B
```

Notice that the 1st and 2nd elements of MTR are FLOAT arrays. If the array MTR is displayed, it would look like this:

```
  2R X 4C
  | 1   2   3    4
--+---------------
1 | 1.  2.  3.   4.
2 | 8.  9.  10.
```

Data within a two-dimensional array is referenced by using two subscripts. Thus in the example above, MTR(1,1) contains the value 1., and element MTR(2,1) contains the value 8. (It may be useful to think of the first subscript of a two-dimensional array as indicating the row and the second subscript as indicating the column of the array.)

Creating Arrays

To create an array, the DECLARE statement (see below, Section IV) must be used. The declaration can take two forms:

```
     DECLARE array-name(*) data-type
 or  DECLARE array-name data-type ARRAY
```

Some examples are:

```
     DECLARE X(*) FLOAT
     DECLARE XX(*,*) FLOAT
     DECLARE XX FLOAT ARRAY ARRAY
     DECLARE CUBE(*,*,*) TEXT
     DECLARE CUBE TEXT ARRAY ARRAY ARRAY
```

Fixed-length Arrays

Note that arrays in PROPHET do not have to be assigned a specific length. It is possible to declare an array with a constant length in PROPHET by the statement:

```
     DECLARE array-name(length) data-type
```

Such a statement, though assigning no values to the array, does say that it consists of a constant number of objects. To take a specific example, the statement

```
     DECLARE XX(10,10) FLOAT
```

says that XX is a two-dimensional FLOAT array consisting of 10 one-dimensional FLOAT arrays, each one of which can accommodate 10 FLOAT values.

Even though you can declare and use constant length arrays in PROPHET, it is strongly suggested that you declare your arrays with an open length (i.e., using asterisks as shown above). The main reason for this is that as procedures are revised, fixed-length arrays have a way of becoming obsolete by outgrowing the amount of space which had been set aside earlier. Another more subtle reason is that when System procedures have an array as an argument, calling them with a constant-length array can cause an error message; this is because PROPHET regards a constant length array as a specific data type, and not as the precise equivalent of an open-ended array of the same type and dimensions, i.e. A(*,*) FLOAT is not the same type as B(*,3) FLOAT or as C(5,3) FLOAT.

Array Constants

As shown above, elements of an array can be assigned a value by referring to subscripts. In addition, an entire array can be assigned values using the "square-bracket" notation:

```
A = [3.14, 6.3, 6.45]
```

There is one distinct limitation to array constants: you cannot assign variables to an array constant. Thus when X, Y, and Z are declared FLOAT variables, you cannot say

```
ARA = [X,Y,Z];
```

In addition, arrays created using an array constant are always variable length arrays. Thus, if you declare A(3) FLOAT, you cannot then assign values to it with an array constant, i.e. the statement

```
A = [1., 2., 3.]
```

will produce an error message.

F. Complex Data Types

Complex data types, as the name implies, are objects which are constructed from combinations of other data types. These constituent parts of a complex data type are called the "Subfields" of the data type. Complex data types can themselves be used as subfields of other complex data types. Thus a GRAPH is a data type which consists of a TEXT (the TITLE subfield of the GRAPH), and three TABLEs (which form the AXES, CURVES, and POINTS subfields of the data type GRAPH).

Referring to Subfields

There are two ways of referring to subfields of complex data types--by using the "DOT" notation, or by using the syntax "OF". Thus the AXES subfield of the graph G may be referred to by the expressions:

```
    G.AXES
or
    AXES OF G
```

Using either expression, you are referring to a TABLE.

Similarly the DEFINITION subfield of a Procedure P may be referred to by the expressions

```
    P.DEFINITION
or  DEFINITION OF P
```

A few complex data types and their subfields are listed below. This is <u>not</u> a complete list of all PROPHET's complex data types, but only of those data types which are most often used in procedures. Some of these data types are discussed in more detail later in this document or in the Users Manual. In addition, for information about PROPHET tables and molecules, you should refer to the the relevant sections of the PROPHET Users Manual.

1. Graphs

The subfields of the data type GRAPH and their corresponding data types are:

TITLE	<text>
AXES	<table>
CURVES	<table>
POINTS	<table>

2. Sample

The subfields and data types of the complex data type SAMPLE are:

TITLE	<text>
DATA	<float array>
MEAN	<float>
STDEV	<float>

3. Procedure

Among the subfields and data types of the complex data type PROCEDURE are:

DEFINITION	<text>
P	<compiled code>

4. Commandlist

The complex data type COMMANDLIST has one subfield

TEXT	<text>

II. Basic P

This section desc
statements permitted
analogous to a sentence
of symbols, or words, w
form a command that
execute.

In general, PL/PROP
any format you may choo
carriage returns or
separate words. At
character is required b
must be terminated wit
";" within procedures,

A. Assignment Statemen

The assignment of v
can be done with the
operator.

```
A = 3
T = '
X = Y
OKFLA
Z = X
```

All of the above are le

The "=" Operator ca
example, while you can

```
A = COLUMN 1 R
```

to assign a specific va

/PROPHET Statements

ibes the basic types of
n PL/PROPHET. A "statement" is
in English; i.e., it is a group
ich when correctly combined,
the computer can understand and

ET statements can be written in
e. Statements can be broken by
ultiple spaces between the
least one space or special
tween words and each statement
a special character--usually a
nd a <GO> otherwise.

s

lues to variables in PL/PROPHET
ET command or with the "="

```
5;
BC';

 = FALSE;
Y;
```

al assignment statements.

not be used in some cases. For
ay

```
 3 OF TAB;
```

ue to A, you cannot say

```
COLUMN 1 ROW 3 OF TAB = A;     /*WRONG!!!*/
```

to assign a specific value to a cell of TAB. In this case you must use the SET command or the %SETCELL procedure (described in Section VII).

```
SET COLUMN 1 ROW 3 OF TAB TO A;

%SETCELL (TAB,3,1,A);
```

The "=" operator is best used for simple variables such as FIXED, FLOAT or TEXT. In assigning values to complex data types such as MOLECULEs, GRAPHs, etc. it is safer to use the SET command.

B. Input/Output Statements

PL/PROPHET has several INPUT/OUTPUT (I/O) statements designed to simplify the transfer of data into and out of procedures. There are some statements which simplify the writing of interactive procedures in addition to passing data into and out of a procedure by means of arguments (described below in Section IV).

1. Output Statements

All of the normal PROPHET display commands are valid within procedures, and they are normally the best way to "output" such variables as TABLEs, GRAPHs, and arrays. For numbers and text, there are some simple statements described here.

The simplest output statement is

```
TYPE variable;
```

The TYPE statement causes the value of "variable" to be output in the command portion of the terminal.

On non-display terminals, a "free" carriage return-line feed is supplied following output with the TYPE or OUTPUT statement. On display terminals, a "free" space is automatically output. When you are outputting TEXT in particular, this can result in badly formatted output. The "free" carriage return-line feed or space can be suppressed by using the statement

```
TYPE NOCR variable;
```

where NOCR stands for "no Carriage-return".

More than one variable can be output with a single TYPE statement. The statement

```
TYPE X1,X2,X3,X4;
```

would cause the values of all four variables to be output with a "free" space or carriage-return separating each value. You can also combine TEXT constants with numeric variables to make the output more intelligible, as in this example:

```
TYPE NOCR 'THE VALUE OF X IS ',X;
```

2. Input Statements

The basic input statement in PL/PROPHET is simply:

```
INPUT variable;
```

This statement causes the program to pause, and the current prompt character (normally set to "?") to be typed. The procedure then waits for the user to respond by typing in a constant or expression or the name of a variable of the correct data type.

To take an example:

```
DECLARE X FLOAT;
statements;
INPUT X;
statements;
```

The execution of this code would cause the following interaction: When the statement INPUT X; is encountered, a "?" would be typed on your terminal. Execution of the program would wait until you typed a FLOAT constant or the name of a FLOAT variable. If you responded by typing "3.4<GO>" the FLOAT variable X in the procedure would be assigned the value 3.4. If you typed the name of a FLOAT variable--%PI for example--the variable X would be assigned the value of that variable, viz. 3.14.

Notice that there is no checking for an EMPTY value being input; thus if you responded to the prompt by typing "A" and there was no FLOAT variable named A, it would be assumed that an EMPTY FLOAT had been input, and X would be correspondingly EMPTY.

The INPUT statement will check the data type of the input variable against the declared type of the procedure's variable. If they are not identical, the error message "ERROR IN INPUT" is typed on the terminal, and you must type in a new value. (This applies to the data types FIXED and FLOAT, so if you want to input a FLOAT variable, you must type a number with a decimal point in response to the prompt.)

Constants and variables of any data type can be transferred into a procedure by means of the INPUT statement. Thus, if the variable "TAB" were declared in a procedure to be a TABLE, the statement INPUT TAB; would require you to type in the name of a table.

The INPUT TEXT Statement

Constants or variables of type TEXT can be input by using the basic INPUT statement, but using that statement has the unfortunate side-effect that when you want to input a TEXT constant, you must enclose it in single quotes. The INPUT TEXT statement is designed to obviate the need of typing the single quotes. Thus the statements

```
DECLARE TXT TEXT;
statements;
INPUT TEXT TXT;
```

would cause the execution of the procedure to pause, the prompt character would be typed on your terminal, and whatever characters you typed would be understood to be TEXT. Thus if you typed "ABC<GO>" the variable TXT would have the value ABC. If you responded to the prompt by typing in "'ABC'<GO>", then the single quotes would be included in the TEXT being input and TXT would have the value 'ABC'.

Note one complication of using the INPUT TEXT statement. If you want to input a TEXT variable, not a TEXT constant, you must precede the name of the variable with the TEXT-KEY, CTRL-R; otherwise the name you type would be assigned as the TEXT you wanted to input. Thus, in the example above, if you wanted to assign to the procedure's variable TXT the value of the TEXT variable %CRLF, you would have to answer the prompt with <TEXT-KEY>%CRLF<GO> to have the desired effect. If you answered the prompt with "%CRLF<GO>", the variable TXT would have the value %CRLF.

Interactive I/O

You will usually want to use a combination of Input/Output statements in order to clarify what variables are being input or output. In the example below, notice that each data item being provided by the user running the procedure is labelled with an identifying TEXT.

```
TYPE NOCR '
PLEASE TYPE IN THE NUMBERS AS PROMPTED
    X1 = ';

INPUT X1;
TYPE NOCR '
    X2 = ';
INPUT X2;

statements;

TYPE NOCR '
HOLLISTER-WOOD''S STATISTIC = ',X3;
```

C. Conditional Statements

A conditional statement is a statement that tests a condition and, based on the results of the test, executes a command. The basic conditional statement in PL/PROPHET is the IF . . . THEN statement and its adjunct, the ELSE statement.

1. The IF . . . THEN Statement

The IF . . . THEN statement can formally be expressed as:

```
IF boolean-expression THEN executable-statement;
```

A "boolean-expression" is simply an expression which when evaluated at runtime is either TRUE or FALSE (see Section I.D.3 above). Here are some examples:

```
IF (X < 3.) THEN . . .;

IF %YESANSWER('DO YOU HAVE A TABLE') THEN . . .;

IF (EMPTY(S)) THEN . . .;

IF (X <> 0.) THEN . . .;

IF (NOT(X = 0.)) THEN . . .;

IF (EXT(1,T) = 'A') THEN . . .;

IF ((X <> 0.) AND (X < 1.)) THEN . . .;
```

The parentheses enclosing the boolean-expression are usually not required, but are useful for depicting the scope of a boolean-expression. Note that such functions as EMPTY and %YESANSWER (see section III) which return a BOOL may be directly used as a boolean-expression.

The right half of an IF . . . THEN statement, the "executable-statement", may be any legal PL/PROPHET statement that can be executed--including a further IF . . . THEN statement. Here are some examples:

```
IF NOTEMPTY(TAB) THEN DISPLAY TAB;

IF (X > 0.) THEN Y = Y + X;

IF OKFLAG THEN TYPE 'CONDITIONS ARE OK';

IF ((X > 0.) AND (X < 1.)) THEN Z = Y/X;
```

Nested IFs

When the "executable-statement" following an IF . . . THEN is another IF . . . THEN, it is called a "Nested IF". An example of a nested IF is

```
IF (X > 0.) THEN IF (X < 1.)  THEN Z = Y/X;
```

Note that the effect of the nested IF is the same as the previous example above. But while the outcome is identical, the execution is slightly different, since <u>both</u> sides of an AND expression are <u>always</u> evaluated. With the Nested IF, when the first test (X > 0.) fails, the second test (X < 1.) will not be made. This is potentially safer and more efficient, and is discussed more fully in Section V below.

2. The ELSE Statement

The ELSE statement is a highly useful variation of the IF . . . THEN statement. An ELSE statement consists of

```
IF boolean-expression THEN  executable-statement;
ELSE  [executable-statement];
```

The optional "executable-statement" following the ELSE can consist of any valid PL/PROPHET statement, including an IF . . . THEN statement, or a DO . . . END block. (see Section II.D. below)

The ELSE statement <u>must</u> immediately follow an IF . . . THEN statement, with no intervening statements. When the boolean-expression of the IF . . . THEN is evaluated as FALSE, the statement following the THEN is skipped and the ELSE statement is executed. Here are some examples:

```
        IF (X < 0.) THEN Y = Y + X;
        ELSE Y = Y - X;
```

When this code is executed, the boolean-expression (X < 0.) is first tested. If, for example, X = -15., the boolean-expression will be TRUE and the statement following the THEN will be executed. If, on the other hand, X = 20., then the statement Y = Y + X will be skipped, and the statement following the ELSE will be executed. This is useful, since you do not have to test for the negative of (X < 0.).

In the following example, an <u>incorrect</u> use of the ELSE statement is shown:

```
    IF (X = 0) THEN Y = Y + X;
    TYPE X;                                     /*WRONG!*/
    ELSE Y = Y - X;
```

What is incorrect here is that the statement TYPE X; immediately follows the IF . . . THEN statement. If the ELSE statement is to be used, it <u>must</u> immediately follow the IF . . . THEN statement. This is a rather frequent error in writing procedures and will result in code that will not compile. Remember that an ELSE statement must "pair" with an IF . . . THEN statement, with no statements logically intervening. (Note in Section II.B below how DO . . . END blocks affect the use of ELSE statements.

Here are some other examples of the ELSE statement:

```
    IF %YESANSWER(' DO YOU WANT TO SCAN THE TABLE')
        THEN SCAN TAB;
        ELSE EAD TAB;
    IF NOTEMPTY(Q) THEN SUMM = SUMM + Q;
    ELSE TYPE 'EMPTY VALUE FOUND';

    IF (T = 'A')
        THEN TYPE 'FOUND AN "A"';
        ELSE IF (T = 'B')
            THEN TYPE 'FOUND A "B"';
            ELSE TYPE 'NO MATCH FOR "A" OR "B" FOUND';
```

Note in the last example how the ELSE in line 3 "pairs" with the IF in line 2, and the ELSE in line 5 "pairs" with the IF in line 4.

```
    IF (T = 'A')

        THEN IF (U = 'B')
            THEN TT = T.U;
            ELSE;

        ELSE TYPE 'NO "AB" MATCH FOUND';
```

Note the need to provide an "empty" ELSE here to match the nested IF. Without the ELSE; in line 2, the ELSE TYPE . . . statement in line 3 would be executed if the IF (U = B) statement were evaluated as FALSE. Since the intention is to execute the TYPE statement when both IFs are found to be FALSE, the "empty" ELSE is needed to "pair" with the inner IF.

D. The DO . . . END Block

The DO . . . END Block is one of the most powerful features of PL/PROPHET. By means of the DO . . . END Block, you can group statements into logical blocks of code and control the flow of commands in a structured way. Thus, if you do an IF . . . THEN statement, and you want to perform more than one operation based on the outcome of the IF . . . THEN test, you can use the DO . . . END statement to group those operations into one logical block. For this purpose, the entire DO . . . END Block is considered to be a single executable statement. For example:

```
IF NOTEMPTY(X) THEN DO;
    TYPE X;
    SUMM = SUMM + X;
    IF (SUMM > 100.) THEN TYPE NOCR 'SUMM = ',SUMM;
END;
ELSE TYPE 'FOUND AN EMPTY';
```

In this, if the boolean test NOTEMPTY(X) in the IF . . . THEN is evaluated as TRUE at runtime, all of the lines of code sandwiched between the DO and the END are executed in sequence. Note also that the ELSE statement on the last line of the example is paired with the <u>first</u> IF . . . END statement, not with the second. The reason for this is that the second IF . . . THEN is wholly contained within the DO . . . END block; it is thus <u>not</u> the preceding logical IF . . . THEN of the ELSE.

A DO . . . END block can also be used following an ELSE statement. The next example is precisely identical in its effect to the one above.

```
IF EMPTY(X) THEN TYPE 'FOUND AN EMPTY';
ELSE DO;
    TYPE X;
    SUMM = SUMM + X;
    IF (SUMM > 100.) THEN TYPE 'SUMM = ',SUMM;
END;
```

Note that in both examples of DO . . . END blocks, we used indentation to clarify the scope of the DO . . . END. This is not required, but it is a good programming practice to follow since it makes it much easier to follow the logic of the program, and also easier to match the END; with the appropriate DO.

DO . . . END Blocks may also be "nested" within other DO . . . END Blocks, as in this example:

```
IF NOTEMPTY(X) THEN DO;
    IF (X <= 50.) THEN DO;
        LOW = LOW + 1;
        SUMLOW = SUMLOW + X;
    END;

    ELSE DO;
        HI = HI + 1;
        SUMHI = SUMHI + X;
    END;

    SUMTOTAL = SUMTOTAL + X;
END;

ELSE EMPTYVALS = EMPTYVALS + 1;
```

Note the following points:

(a) The ELSE in the last line of the example is paired with the IF . . . THEN of the very first line. The ELSE in line 6 is matched with the IF . . . THEN in line 2.

(b) The statement SUMTOTAL = SUMTOTAL + 1 will be executed if the condition in the first line is met. The further IF . . . THEN statement in line 2 and the ELSE in line 6 have no effect on it.

E. The DO Loop

The DO Loop is an extension of the DO . . . END Block which allows a group of statements to be executed in an iterative, or repetitive way.

1. The Basic DO Loop

Formally the syntax for the basic DO Loop is:

```
DO counter = start-value TO quit-value [BY increment];
   statements;
END;
```

where "counter", is normally a numeric variable; "start-value", "quit-value", and "increment" are typically numeric variables, constants or expressions. The "BY increment" is optional, and allows you to increase (or decrease) the "counter" for each iteration by a value other than the default value of 1. The DO Loop must be closed with an END.

The formal mechanics of a DO Loop are:

(a) Before the initial iteration, "counter" is assigned the value of "start-value".

(b) Before each iteration after the first, the current value of "counter" is increased (or decreased) by the value of "increment" (which is always 1, unless the "BY increment" statement specifies another value).

(c) For each iteration (including the first), the current value of "counter" is compared to the value of "quit-value". If "counter" is less than or equal to the "quit-value", the statements within the DO Loop are executed. If, on the other hand, "counter" is greater than "quit-value", the iteration is terminated immediately and the flow of control jumps out of the DO Loop to the statements following the END of the loop. (If "increment" is negative, i.e. a decrement, "quit-value" is less than "start-value" and these tests are reversed.)

(d) The statements enclosed by the DO and the END lines of the DO Loop are executed in sequence. When the END; of the DO Loop is encountered, control jumps back to the beginning of the loop and Step (b) is carried out.

Here is a simple example of a DO Loop.

```
DO COUNTER = 1 TO 6;
    TYPE 'PASS';
    TYPE COUNTER;
END;
```

When this code is executed, the variable COUNTER will initially be assigned the value 1; it will be compared to the "quit-value" of 6. On the first iteration, 1 is less than 6, so the statements between line 1 and line 6 will be executed in sequence and PASS 1 is typed out. When the END is encountered, control jumps back to the first line. COUNTER is increased by 1 (since no "BY increment" specification is given). COUNTER is still less than 6 so the statements are executed again and PASS 2 is typed. This continues until after the sixth iteration. When control has returned to the first line, COUNTER is incremented by 1. When the comparison of COUNTER and "quit-value" is made, it is seen that 7 is greater than 6, so control will jump out of the loop, and continue with the statements following the END. By this time the typed output looks like this;

```
PASS 1 PASS 2 PASS 3 PASS 4 PASS 5 PASS 6
```

Note that the value of COUNTER will be 7 after this loop is completed. Thus, when a DO Loop using the default increment value of 1 terminates, the counter variable is normally equal to the value of "quit-value" plus 1.

Some further examples of DO Loops using a non-default increment are shown below. You might try these examples, and some of your own, at top-level PROPHET to get a better idea about the execution of DO Loops.

```
DO I = 100 TO 1 BY -1;
    TYPE I;
END;
```

Here, the use of the "BY -1" would cause the TYPE statement to output:

```
100 99 98 97 96 . . . 5 4 3 2 1
```

The counter variable, I would have the value 0 when the loop is completed.

```
DO I = 1 TO 99 BY 2;
    TYPE I;
END;
```

Here, the TYPE statement would be executed exactly 49 times, and the output would be:

```
1 3 5 7 9 11 . . . 91 93 95 97 99
```

In this example the counter variable would have the value 101 when the loop is completed.

```
DO I = 0 TO 10 BY 12;
    TYPE I;
END;
```

Here, the TYPE statement would be executed only once, since on the second iteration, the value of I would be greater than the "quit-value" of 10. The only output would be "0" and the value of I after the loop is completed would be 12.

Nested DO Loops

A DO Loop within another DO Loop is called a "Nested DO Loop". Nested DO Loops are frequently used to perform operations on PROPHET tables, or on two-dimensional arrays. In the example below, the sum of the "rows" of a two-dimensional FLOAT array are computed.

```
DO I = 1 TO TOP(ARA);
    ROWSUM = 0.;

    DO J = 1 TO TOP(ARA(I));
        ROWSUM = ROWSUM + ARA(I,J);
    END;

    TYPE ROWSUM;
END;
```

Note in line 1 and line 3 that we can use functions which return values in the DO Loop statement. Thus in line 1 the TOP function (described below in Section III) can be used in place of a FIXED constant as the "quit-value" for the DO Loop. Note also the use of indentation to clarify the scope of the two DO Loops. Finally, it is important to note that the values of TOP(ARA) and TOP(ARA(I)) are evaluated each time through the DO Loops. So you should be careful not to do anything within the loops which changes the size of ARA or the DO Loops may have surprising results.

Alternate Constructions of DO Loops

Another possible construction of the DO Loop statement is

```
DO counter = value1, value2, value3, etc.;
   statements;
END;
```

Some examples of this type of DO Loop are:

```
DO I = 1,5,7,17,55;
    TYPE ARA(I);
END;
DO T = %CRLF,%TAB,' ',%FF;
    IF (U = T) THEN TYPE 'FOUND '.U;
END;
```

Note that non-numeric variables <u>can</u> be used in DO Loops, although numeric counters are much more typical.

It is also permitted to combine the two forms of the DO Loop, as in this example.

```
DO I = 1 TO 5,15,20,25 TO 27;
    TYPE ARA(I);
END;
```

Here, the counter, I would take on the values

```
1 2 3 4 5 15 20 25 26 27
```

Finally, you can use just a "start-value" and a counter, e.g.

```
DO I=1 BY 1;
```

but you must provide some other way to quit executing the DO Loop. Some of these are discussed below in Sections II.E.2, II.E.3 and II.F.

Guidelines in Using DO Loops

A few suggested guidelines when using DO Loops are worth remembering:

(a) Never change the value of the counter variable within the DO Loop. If you increase the value of the counter you will probably exit prematurely from the DO Loop. Even more serious, if you decrease the counter within the DO Loop, you may create an infinite loop, i.e., a DO Loop that never terminates. Here is a simple and lethal example:

```
DO I = 1 TO 5;
    I = I - 1;       /*   WRONG!!!  */
    TYPE I;
END;
```

In this example, as fast as the value of I is incremented, line 2 decreases it. The counter value will <u>never</u> reach a value greater than 5 and the Loop will continue executing (and outputting an infinite stream of "0 0 0 0 0 . . . ").

(b) A second thing to remember when using DO Loops is that the variables used for "counter", "start-value", and "quit-value" should be of the same data type. While no error will result if you mix FLOAT and FIXED variables, the execution of the code will be much slower than if the variables were all of the same data type. (Also, roundoff errors may cause the Loop to execute one too few or one too many times.)

(c) It is generally not a good programming practice to rely on the counter variable to have reached a certain value after the execution of a DO Loop. While it is certainly true that the counter will be equal to the "quit-value" plus 1, in the "normal" case of a DO Loop which uses the default value for the increment, you can see from the examples above, that the "counter" variable can have some quite divergent values.

2. DOEXIT and DONEXT

The DOEXIT and DONEXT statements are designed to allow greater flexibility in the execution of DO Loops.

The DOEXIT Statement

When the DOEXIT statement is encountered in a DO Loop, the flow of control immediately exits from the loop and the statements following the closing END; of the loop are executed. The DOEXIT statement can appear as the "executable-statement" of an IF . . . THEN statement as in the example below:

```
DO I = 1 TO Z;
    ARA(I) = MTR1(I) * MTR2(I);
    IF (ARA(I) >= 1000.) THEN DOEXIT;
END;

DISPLAY ARA;
```

When the boolean-expression (ARA(I) >= 1000.) is evaluated as TRUE, the DOEXIT statement is immediately executed, and control shifts to the DISPLAY ARA statement following the closing END of the DO Loop. If the boolean-expression is not evaluated as TRUE before the counter I reaches the "quit-value" of Z, then the DOEXIT statement will not be executed and the loop will terminate normally.

The DONEXT Statement

When the DONEXT statement is encountered in a DO Loop, the flow of control is immediately sent back to the first line of the DO Loop and the next iteration takes place. In the example below, note how the use of DONEXT allows some of the statements within a DO Loop to be "skipped" over.

```
DO I = 1 TO Z;
    SUMM = SUMM + ARA(I);
    IF (SUMM <= 1.) THEN DONEXT;
    SUMM = SUMM + .035;
END;
```

When the boolean-expression (SUMM <= 1.) is evaluated as TRUE, control will shift back to the first line of the DO Loop, DO I = 1 TO Z and the next iteration will occur. If the boolean-expression is evaluated as FALSE, then the last statement in the DO Loop, SUMM = SUMM + .035 will be executed.

It is important to remember that both DOEXIT and DONEXT operate only within DO Loops, not within DO . . . END Blocks.

DOEXIT and DONEXT operate only for the innermost DO Loop in which they occur. Thus, in nested DO Loops, when a DONEXT or DOEXIT statement is contained in the code of the innermost DO Loop, the execution of the DONEXT or DOEXIT will be for that innermost loop, not for the outer loop.

```
DO I = 1 TO NR;
    IF EMPTY(TARA(I,1)) THEN DOEXIT;  /* DOEXIT 1 */
    SUMM = 0.;

    DO J = 1 TO NC;
        SUMM = SUMM + TARA(I,J);
        IF (SUMM >= 100.) THEN DOEXIT; /*DOEXIT 2 */
    END;

    TYPE SUMM;
END;
```

Note that if the first DOEXIT is executed, control would jump out of the outermost DO I Loop. If the second DOEXIT is executed, control would jump out of the DO J loop only, and iteration of the DO I Loop would continue. If you want to get all the way out, you could use GO TO statement discussed in Section II.F below).

3. The WHILE Extension of the DO Loop

A further extension of the basic DO Loop statement is the WHILE statement. In its simplest form, it is:

```
DO WHILE (boolean-expression);
    statements;
END;
```

where "boolean-expression" is an expression which when evaluated is either TRUE or FALSE, and the parentheses are requied in this case. An example of the WHILE extension is:

```
A = 0;
B = 2;
DO WHILE (A <> B);
    A = A + 1;
    TYPE A;
END;
```

The loop will continue iterating until control returns to the first line and the boolean-expression (A <> B) is evaluated as FALSE. It is important to realize that the boolean-expression following the WHILE is evaluated only when control has gone back to the first line of the DO Loop. When the A = A + 1 assignment statement results in A being assigned the value of 2, the TYPE A statement will still be executed; exit from the loop occurs only when the control returns back to the first line and the test (A <> B) is evaluated as FALSE.

The WHILE extension can be used in conjunction with a "counting" DO Loop. This makes it possible to end the iteration based <u>either</u> on the value of the counter, or on the result of the WHILE test. To take an example:

```
Y = 1000;
X = 0;

DO I = 1 TO TOP(ARA) WHILE (X < Y);
    X = X + ARA(I);
END;
```

If the array contained the numbers [500,400,95], the loop would iterate 3 times, since the value of X would not become greater than Y. On the other hand, if the array contained the numbers [900,101,1,1,3,6000], iteration would cease after 2 loops since the WHILE condition would no longer be met (X would equal 1001).

4. DO FOREVER

The DO FOREVER statement is a convenient way of setting up an unterminated DO Loop. Since it is an unterminated, i.e., an infinite, loop, an explicit exit from a DO FOREVER loop <u>must</u> be provided--either by a DOEXIT, by a GO TO statement (see Section II.F) or by a RETURN statement (see Section II.G).

A DO FOREVER loop is particularly useful in writing an interactive data input procedure, as in the example below.

```
TYPE '
PLEASE TYPE IN THE TABLES NAMES AS PROMPTED.
  WHEN YOU ARE DONE TYPE "EXIT"';
DO FOREVER;
    TYPE '
  TABLE NAME';
    INPUT TEXT T;
    IF (T = 'EXIT') THEN DOEXIT;
    ELSE PRINT CTABLE(T);
END;
```

The DO FOREVER Loop will continue iterating until you answer the prompt by typing in "EXIT". When you input that text, the boolean-expression (T = 'EXIT') will be evaluated as TRUE, and the DOEXIT will be executed. Note, too, that there are several other ways of writing equivalent code. The point to remember is that DO Loops in PL/PROPHET provide a great deal of flexibility.

F. The GO TO Statement and LABELs

In addition to controlling the flow of control in a procedure by means of logical blocks formed by DO. . . . END blocks, control can also be directed by using the GO TO statement and LABELs. The form of the GO TO Statement is:

```
GO TO label-name;
statements;

label-name: statements;
```

When the flow of control reaches the GO TO statement, control is shifted from the statements following the GO TO and is directed to the LABEL named by the GO TO. Note that a LABEL is indicated in the code by a colon ":" following it. The LABEL can precede or follow a GO TO which references it. The following example shows a typical use of LABELs and GO TO statements.

```
I = 0;
ASKIT:
I = I + 1;

INPUT ARA(I);
IF %YESANSWER('MORE NUMBERS') THEN GO TO ASKIT;

DISPLAY ARA;
```

As long as the condition following the IF is true; i.e., as long as the user keeps answering the question "MORE NUMBERS?" with a "YES", the program will continue to go back to the label ASKIT, and execute the statements between the LABEL and the GO TO. The statement DISPLAY ARA; will not be executed until the GO TO ASKIT is not executed; i.e., until the user answers the question with a "NO". Note that a GO TO statement can form the "executable-statement" of an IF . . . THEN statement.

1. LABELs

LABELs are a data type in PL/PROPHET, and must always have legal PL/PROPHET names: i.e., they must begin with an alphabetic character, contain only alpha-numeric characters, and have no spaces. LABELs must always be followed by a colon ":".

No label name may appear more than once in a procedure. If you want to use the same label names in a subprocedure, e.g.

```
DECLARE LAB LABEL;
```

and in the parent procedure, you must DECLARE the labels in the subprocedure.

2. Limitations on GO TOs

You should never use a GO TO to jump into the middle of a DO . . . END block, or into the middle of a DO Loop.

3. Caveats on Using GO TOs

GO TOs should generally be avoided in PL/PROPHET. There are several reasons for this:

(a) Heavy use of GO TOs and LABELs tends to produce code that is extremely difficult to follow and maintain.

(b) Using DO . . . END blocks and subprocedures in lieu of GO TOs is both easier to follow and much more efficient in execution. In the examples below are illustrations of how DO . . . END blocks and DO Loops can be used instead of GO TOs.

The first example shows the example above could be rewritten using block structure tools.

```
DO I = 1 BY 1;
    INPUT ARA(I);
    IF NOT(%YESANSWER('MORE NUMBERS'))
      THEN DOEXIT;
END;
DISPLAY ARA;
```

Note that this takes fewer lines than the equivalent code above and does not need the LABEL.

```
IF (T = 'A') THEN GO TO L1;
ELSE IF (T = 'B') THEN GO TO L2;
ELSE GO TO L3;
L1:  /* T = 'A' */
TYPE 'FOUND A';
U = U.T;
GO TO L4;
L2:  /* T = 'B' */
TYPE NOCR 'FOUND B';
V = V.T;
GO TO L4;
L3: /* T <> 'A' OR 'B' */
TYPE 'A OR B NOT FOUND';
L4: /*PROGRAM CONTINUES */
```

Contrast the code above with the code below, which uses the block structure tools of PL/PROPHET:

```
IF (T = 'A') THEN DO;
    TYPE 'FOUND A';
    U = U.T;
END;
ELSE IF (T = 'B') THEN DO;
    TYPE 'FOUND B';
    V = V.T;
END;
ELSE TYPE 'A OR B NOT FOUND';
```

The code in both examples will have the same effect when executed (though the first will be slightly slower); but look how much easier it is to follow the logic of the second. In addition, the code for the first example will take up slightly more space than the code for the second.

(c) You should take special care with GO TOs when converting FORTRAN code to PL/PROPHET. It is much better to restructure the code rather than merely convert the program line-for-line. The extra time it will take to do this will be amply repaid later when you need to modify the program.

G. The RETURN Statement

The RETURN statement has two distinct forms and uses. When it is used in the form

```
RETURN;
```

the effect is to immediately exit from the procedure being executed and return to the place from which that procedure was called.

A RETURN statement can appear by itself or as the "executable-statement" in an IF . . . THEN statement. Thus in the following example:

```
statements;
IF (X < 1) THEN RETURN;
statements;
```

When the boolean-expression (X < 1) is evaluated as TRUE, an immediate exit from the procedure occurs and control is returned to the location from which was called. That is, if it was called from top-level PROPHET, the execution of the RETURN statement would return control to top-level PROPHET; if it had been called from another procedure, control would be returned to that procedure and the statements following the call would be executed. If a subprocedure contains a RETURN statement, and the RETURN is executed, control passes back to the parent procedure which called the subprocedure.

The RETURN statement can also be used to transfer data from a procedure. This form of the RETURN statement is discussed in Section IV.

H. The CLEAR Statement

The statement

```
CLEAR variable;
```

is designed to wipe out the current contents of "variable" and set it to EMPTY. The data type of the variable is unchanged.

The CLEAR statement is particularly useful when you want to use the same variable within a procedure for different purposes in different sections of the procedure. Each time you are finished using the variable, you can use the CLEAR statement to wipe out its current contents, and then proceed to use it again.

When you are using particularly large variables -- e.g., GRAPHs, MOLECULEs, or arrays -- it is often useful to use the CLEAR statement when you are finished with them in order to recover some of the space they occupy.

When you want to recover space used by permanent TABLEs within a procedure, you should use the PARE command discussed in Section II of the User's Manual. Never use the CLEAR statement with permanent TABLEs.

III. Basic PL/PROPHET Functions and Procedures

PL/PROPHET has built-in functions for most common mathematical operations, as well as functions designed to manipulate textual data, to convert data from one type to another, and to assist the writing of interactive dialogue.

A FUNCTION is simply a procedure which returns a value. A function can consequently be used directly in any PL/PROPHET statement wherever a constant or a variable of the data type returned by the function would be allowed.

A. The EMPTY and NOTEMPTY Functions

The functions EMPTY and NOTEMPTY are used to test if a variable has any value assigned to it or not. Each function takes a single argument, the variable to be tested, and each returns a BOOL. Table 3.1 depicts the form of these two functions.

EMPTY variables may be used to form relational expressions. Attempts to perform any other kinds of operations using an EMPTY variable will invariably result in the error message

```
128. Undefined Variable Reference
```

Therefore, it is always wise to use the EMPTY or NOTEMPTY functions on variables whose status may be questionable, e.g. individual elements of an array.

Table 3.1 EMPTY & NOTEMPTY FUNCTIONS

Function	Type of Argument	Type of Result	Explanation
EMPTY(variable-name)	(any type)	BOOL	Returns TRUE if the variable named is EMPTY, and FALSE otherwise.
NOTEMPTY(variable-name)	(any type)	BOOL	Returns TRUE if the variable named is not EMPTY and FALSE otherwise.

B. Data Conversion Functions

It is sometimes desirable to convert data from one type to another. For example if you wanted to combine TEXT with FIXEDs or FLOATs to form neatly formated output from a statistical procedure, it would be desirable to turn the numeric data into textual data and then use the concatenation operator to combine the pieces. The data conversion functions allow this type of operation. Table 3.2 shows the basic PL/PROPHET data conversion functions.

Table 3.2 DATA TYPE CONVERSION FUNCTIONS

Function	Type of Argument	Type of Result	Description
CFIXED(arg)	FIXED	FIXED	No conversion is done. The original value of "arg" is the value of the function.
	FLOAT	FIXED	Rounds "arg" to the nearest integer. "arg" can be any valid FLOAT expression. If the magnitude is outside the range of representable FIXEDs then 0 is usually returned. Examples: CFIXED(43.5) returns 44 CFIXED(A+B) where A=12.4 and B=-6.18 returns 6.
	TEXT	FIXED	If the text can be interpreted as an integer, it is converted to integral representation. Alphabetic strings are not valid. An error condition occurs for strings representing integers outside the range of representable FIXEDs. Examples: CFIXED(A) where A is the text '234' returns 234. If A is the text '12.3', then CFIXED('12.3') returns 12.

Table 3.2 DATA TYPE CONVERSION FUNCTIONS (continued)

Function	Type of Argument	Type of Result	Description
CFLOAT(arg)	FIXED	FLOAT	Converts "arg" to the equivalent FLOAT. "Arg" can be any valid FIXED expression. For example: CFLOAT(4300000) returns 4.3E6 and CFLOAT(1) returns 1.0
	FLOAT	FLOAT	No conversion is done. The original value of "arg" is the value of the function.
	TEXT	FLOAT	If the TEXT can be interpreted as a number, it is converted to FLOAT representation. Alphabetic texts are not valid (except for the E used in exponential notation). An error condition occurs for texts representing values outside the range of representable FLOATs.
CTEXT(arg)	FIXED FLOAT TEXT	TEXT	CTEXT converts "arg" to the TEXT representation of the format which would be printed by the TYPE or OUTPUT statement.

C. The COPY Function

The COPY function has a very simple purpose. It returns an exact copy of the argument-variable. The form is

```
new-variable = COPY(old-variable)
```

"New-variable" and "old-variable" should be of the same data type. This is useful when a variable of a complex data-type such as MOLECULE or GRAPH is involved. If you simply say

```
new-variable = old-variable
```

you have only defined another name by which the data in the complex variable can be accessed. If you now change a part of "new-variable", "old-variable" will also be changed. If you use the COPY function, however, "new-variable" is distinct from "old-variable" and changes to "new-variable" will not be reflected in "old-variable".

D. Arithmetic Functions

Table 3.3 shows the basic arithmetic functions available in PL/PROPHET, describing the type of data each returns, the type of the input argument(s), and any limitations. Note that many of these functions will accept either a FLOAT argument or a FIXED argument and that the type of the result may depend on the type of the argument the function was called with.

Table 3.3 ARITHMETIC FUNCTIONS

Function	Type of Argument	Type of Result	Description
ABS(x)	FLOAT	FLOAT	Absolute value of x
	FIXED	FIXED	
ACOS(x)	FLOAT or FIXED	FLOAT	Arccosine of x in radians
ASIN(x)	FLOAT or FIXED	FLOAT	Arcsine of x in radians
ATAN(x)	FLOAT or FIXED	FLOAT	Arctangent of x in radians $-\pi/2 < ATAN\ (x) < \pi/2$
CEIL(number)	FLOAT	FIXED	Returns the smallest integer which is greater than or equal to "number". For example, if RATE = 3.17 and XVALU = 27.0 then 2.*CEIL(RATE) is 8 and CEIL(XVALU) is 27.
COS(x)	FLOAT or FIXED	FLOAT	Cosine of x (x in radians), $\lvert x\rvert<100$.
COSD(x)	FLOAT or FIXED	FLOAT	Cosine of x (x in degrees)
EXP(x)	FLOAT or FIXED	FLOAT	"e" raised to the power "x"; EXP(x) < = 1.701412E8 i.e. x<88.0296922)
FLOOR(number)	FLOAT	FIXED	Returns the largest integer less than or equal to "number". For example, if RATE = 3.17 and XVALU = 27.0 then TYPE 2.** FLOOR(RATE) 8.0 TYPE FLOOR(XVALU) 27

Table 3.3 ARITHMETIC FUNCTIONS (continued)

Function	Type of Argument	Type of Result	Description
LOG(x)	FLOAT or FIXED	FLOAT	Natural logarithm of x, x>0.
LOG10(x)	FLOAT or FIXED	FLOAT	Common logarithm of x, x>0.
MAX(arg1,...arg n)	*(note 1)	*(note 1)	Chooses the largest value from among the arguments.
MIN(arg1,...arg n)	*(note 1)	*(note 1)	Chooses the smallest value from among the arguments.

MOD(number,modulus)	number FIXED
	modulus FIXED
		FIXED	Returns the integer remainder of "number" // "modulus". For example, the following statement will branch to "EVEN" if COUNT is even, and branch to "ODD" if COUNT is odd:
			`IF MOD(COUNT,2)=0`
			`THEN GO TO EVEN:`
			`ELSE GO TO ODD;`

RANDOM	(none)	FLOAT	Returns a pseudo-random FLOAT number uniformly distributed between 0.0 and 1.0. (The pseudo-random number generator repeats itself after 2**27 cycles.)

*Note 1 Arguments can be FIXED or FLOAT. Result will be FIXED if all arguments are FIXED, and will be FLOAT otherwise.

Table 3.3 ARITHMETIC FUNCTIONS (continued)

Function	Type of Argument	Type of Result	Description
%RANINIT [(initial-value)]	FIXED	FIXED	Initializes the pseudo-random number generator and returns the number used for initialization. If the optional parameter "initial-value" is given, the bottom 27 bits are used to initialize the generator. If no "initial-value" is specified, then the time of day in milliseconds is used for initialization. Two sequences of numbers generated from the same initial value are the same. If you use %RANINIT with no argument, then it is a good idea to save the returned initial value somewhere so that you can get the same sequence (e.g., for debugging) later.
%ROUNDOFF (arg, power)	arg FLOAT power FIXED	FLOAT	Returns "arg" rounded to "power" of 10. %ROUNDOFF (12.38,-1) returns 12.4; %ROUNDOFF (182.3,1) returns 180.
%SIGFIG (arg, sigfigs)	arg FLOAT sigfigs FIXED	FLOAT	Returns "sigfigs" significant figures of "arg". %SIGFIG (.03218,3) returns .0322; %SIGFIG (13578.2,4) returns 13580.
SIN(x)	FLOAT or FIXED	FLOAT	Sine of x (x in radians), $\|x\|<100$.
SIND(x)	FLOAT or FIXED	FLOAT	Sine of x (x in degrees)
SQRT(x)	FLOAT or FIXED	FLOAT	Square root of x, x => 0.

E. Text Functions

The basic TEXT functions are shown in Table 3.4.

Table 3.4 TEXT FUNCTIONS

Function	Type of Argument	Type of Result	Explanation
CHARS(string)	TEXT	FIXED	Returns the number of characters in "string". If "string" is EMPTY or is the null text then CHARS returns 0.
EXT(first,[last,]text)	first FIXED last FIXED text TEXT	TEXT	Returns the value of the TEXT between the characters number "first" and "last", inclusive. For example, if MSG is the text 'ABCDEABCD' then EXT(3,7,MSG) returns "CDEAB'. If "last" is not specified, then the value returned is the single character numbered "first". If "first" > "last" then the text returned is the null text (the text with no characters). If "first" < 1 then it is treated as though it were 1. If "last" is greater than the number of characters in "text" then it is treated as though it were equal to the number of characters in "text". Also, if LAST = -1, it is treated as though it were equal to the number of characters in "text".

Table 3.4 TEXT FUNCTIONS (continued)

Functions	Type of Argument	Type of Result	Explanation
LOC(string1,string2 [,start-char])	string1 TEXT string2 TEXT start-char <fixed>	FIXED	Checks to see if "string1" occurs in "string2". If it does not, then the value of the function is 0. If a match occurs, the value of the function is the position (i.e., the number of the first character) of the substring of the second text that matches the first. For example, if BUFFER contains the text 'REQUEST FILE NUMBER' and PEG the text 'FILE', then the statement `POINTER=LOC(PEG,BUFFER)` would set the variable POINTER equal to 9. (The optional third argument may be used to start searching for occurences of "string1" in "string2" beginning with the "start-char" character from the beginning of "string2".)

F. Type-Checking Functions

The functions shown in Table 3.5 are used to determine the data type of a variable. Note that the TYPE function can be used with any variable of any type, while TYPETEST can only be used to test the current type of variables originally declared to be of type ANY.

Table 3.5 TYPE-CHECKING FUNCTIONS

Function	Type of Argument	Type of Result	Description
TYPE [OF] (expression)	(any type)	TEXT	Returns a TEXT which is the name of the data type of "expression" at execution time. If PROPHET has no knowledge of a variable named in "expression", TYPE will return 'NULL'. If a variable in "expression" has been used and then deleted, TYPE will return 'USED IN A PATT'.
TYPETEST(variable-name, data-type)	variable ANY data-type PATT	BOOL	Returns TRUE if at run time "variable-name" is of type "data-type" and FALSE otherwise. (Note that "data-type" should not be in quotes and the variable named must originally have been declared of type ANY.)

G. Input/Output Functions

The functions described in this section are designed to aid in writing interactive programs. In addition to the I/O functions described in this section, there are others described in Section 13 below.

1. %YESANSWER

The %YESANSWER function has a single argument, a TEXT which is output on the terminal. The program then waits for the user to answer the question which has been output. Only two responses will be allowed: "YES" (or "Y") and "NO" ("N"). %YESANSWER returns a boolean value, TRUE if the user answers "YES" and FALSE if "NO". If the user answers anything else, the function asks the question again. The calling sequence is:

```
     answer-flag = %YESANSWER(query)
       <bool>                 <text>
```

2. %GETFIXED

%GETFIXED is a function which outputs a question on the terminal and waits for you to type in a constant or variable of type FIXED which is returned by the function. It has the additional option of allowing you to specify whether you will allow an EMPTY FIXED to be returned or not. The calling sequence of %GETFIXED is:

```
     fix-val = %GETFIXED(query,empty-flag)
     <fixed>             <text>   <bool>
```

where "query" is a TEXT question to be typed out, and "empty-flag" is a BOOL which, if set to TRUE, will allow an EMPTY FIXED value to be returned, e.g. if you type a CTRL-S (the EMPTY key) in response to the prompt. If "empty-flag" is set to FALSE, an EMPTY will *not* be accepted and you will be asked to supply another FIXED value. If %GETFIXED is called with an empty second argument, it will be assumed that the second argument is FALSE and an EMPTY value will not be allowed; i.e., %GETFIXED(query,) is the equivalent of %GETFIXED(query,FALSE).

3. %GETFLOAT

%GETFLOAT is analogous to %GETFIXED, and returns a FLOAT variable or constant input by the user after a question is typed. The "empty-flag" argument is handled as in %GETFIXED.

```
float-val = %GETFLOAT(query,empty-flag)
 <float>               <text>   <bool>
```

4. %GETTEXT

%GETTEXT is similar to %GETFIXED and %GETFLOAT and returns a TEXT which is input by the user after a question is typed on the terminal. A slight difference is that if the "empty-flag" argument is set to TRUE, %GETTEXT will accept and return *either* an EMPTY TEXT value (if you type CTRL-S) or a *null* string (if you type <GO>).

```
text-val = %GETTEXT(query,empty-flag)
 <text>              <text>   <bool>
```

5. %GETANSWER

%GETANSWER is somewhat more complicated, allowing you to respond to a question which is output on the terminal, but limiting the response to a pre-selected list of options. Its calling sequence is

```
ans = %GETANSWER(query, ans-array, num-of-answers,sig-chars)
<text>            <text> <text-array>    <fixed>      <fixed>
```

where "query" is a TEXT question to be asked; "ans-array" is a TEXT array of the acceptable answers; "num-of-answers" is the FIXED number of acceptable answers; and "sig-chars" is the FIXED number of characters that are "significant," i.e., the number of characters which will be accepted as a unique match.

%GETANSWER might be used like this:

```
 TABOPT = %GETANSWER('
ANALYSE BY ROWS (R) OR COLS (C)',['COLUMNS','ROWS'],2,1);
```

The question "ANALYSE BY ROWS (R) OR COLS (C)" will be typed out and you will be prompted for an answer. If you answer with "ROWS" or "R" (or with any word whose first letter is "R"), TABOPT will equal 'R', since in this example "sig-chars" = 1. If you type anything except a word beginning with an "R" or a "C," then %GETANSWER will type out an error message and prompt you for another answer.

Note that the TEXT returned by %GETANSWER is equal to the shorter version of the answers in "ans-array"; i.e., if "sig-chars" equals 1, it will be the first letter. If an exact match is desired, then the last argument can be left empty; e.g.,

```
STATOPT = %GETANSWER('
WHICH STATISTIC',['MEAN','MEDIAN'],2,);
```

You must type either "MEAN" or "MEDIAN"; no other answer will be accepted.

If you want to allow an EMPTY answer, you should set the argument "num-of-answers" to 1 more than the number of answers actually contained in "ans-array".

H. Array Functions

Table 3.6 shows the basic functions used with arrays.

Table 3.6 ARRAY FUNCTIONS

Function	Type of Argument	Type of Result	Description
%ARRAYLOC(seek,array [,start-index])	seek ANY array ANY start-index FIXED		%ARRAYLOC is similar to LOC, except that it looks for any type of data in an array and returns the index of the first array element which contains the datum. The optional third argument allows you to begin the search at an element other than the normal 1.
%ARRAYTEST(expression)	(any type)	BOOL	Returns TRUE if at run time "expression" is an array, and FALSE otherwise.
%BASICTYPE(expression)	(any type)	TEXT	Returns a TEXT which is the base type of the argument, i.e. the data type of data in the lowest-level array. For example, if TEMPS is a FIXED ARRAY ARRAY, then %BASICTYPE(TEMPS) would return the TEXT 'FIXED'. If "expression" is not an array, then the value returned is the same as the value which would be returned by the function TYPE.

Table 3.6 ARRAY FUNCTIONS (continued)

Function	Type of Argument	Type of Result	Description
%ENTRYTYPE(expression)	(an array of any type)	TEXT	Returns a TEXT which is the type of an entry in "expression". For example, if TEMPS is a FIXED ARRAY ARRAY, then %ENTRYTYPE(TEMPS) would return the TEXT 'FIXED ARRAY'. If "expression" is not an array, then an error message is printed.
TOP(array-name)	(an array of any type)	FIXED	Returns the subscript of the last non-empty element of the array, and zero if the entire array is empty.

I. Date and Time Functions

Table 3.7 shows some of the more useful functions which return information about the time and date. In all cases, local time in Boston (if relevant) is used.

Table 3.7 DATE/TIME FUNCTIONS

Function	Type of Result	Explanation
DATE	TEXT	Returns the date according to the computer. It has the form MM/DD/YY where MM, DD, and YY are numbers representing the month, day, and year respectively.
TIME	FIXED	Returns the time of day (measured from midnight) in seconds, read from the time-of-day clock in the computer.
RUNTIME	FLOAT	Returns the number of seconds of CPU time used since the job was logged in.
REALTIME	FLOAT	Returns the number of seconds of real time since the job was logged in, according to the real time clock in the computer.

J. System Variables

In addition to the system functions described above, there are some system variables which may be useful. These are shown in Table 3.8.

Table 3.8 SYSTEM VARIABLES

Name	Type	Value
%PI	FLOAT	3.141593
%E	FLOAT	2.718282
%MAXPOSFLOAT	FLOAT	1.701412 * 10**38
%MINPOSFLOAT	FLOAT	1.469368 * 10**-39
%MAXNEGFLOAT	FLOAT	-1.701412 * 10**38
%MINNEGFLOAT	FLOAT	-1.469368 * 10**-39
%MAXFIXED	FIXED	34359738367 (= 2^{35}-1)
%MINFIXED	FIXED	-34359738378 (= -2^{35})
%CRLF	TEXT	carriage-return line-feed
%FF	TEXT	a form feed (CTRL-L)
%TAB	TEXT	the horizontal tab
%GO	TEXT	the <GO> character
%DISPLAYSWITCH	BOOL	TRUE, if a display terminal; otherwise FALSE

IV. Procedures in PL/PROPHET

A PL/PROPHET Procedure consists of a DEFINITION -- a TEXT which contains a structured set of statements and commands -- and COMPILED CODE produced by the compiler from the definition, which is executed when you use the CALL command.

In the following sections we will describe these components of a procedure in some detail.

A. The Definition of a Procedure

The DEFINITION of a procedure is a subfield of type TEXT containing a structured set of PL/PROPHET statements and commands. This is sometimes referred to as the "source code". The following sections describe the structure of a procedure definition.

A typical example of a PL/PROPHET procedure definition is shown in Figure 4.1 and will be referred to often in the ensuing discussion.

1. Comments in a Procedure

COMMENTS are statements within the definition of a procedure which are neither compiled nor executed. They are simply what their name suggests, comments on what a procedure is intended to do. Comments are set off from the statements of the procedures by special characters. They begin with the characters "/*" (slash, asterisk) and end with the characters "*/". You may include any characters within these comment delimiters except "/*" or "*/". Comments may continue through any number of lines. They may be placed anywhere within a procedure's definition except in the middle of a single word.

```
/* This procedure turns a MOLECULE in 30 degree
   increments around the X-Axis and offers a
   chance to make a copy */


MOLTURN: PROCEDURE(MYMOL);       /* Opening line */

DECLARE MYMOL MOLECULE;          /* The molecule to be turned */

DECLARE I FIXED;                 /* DO Loop counter  */

EAD MYMOL;                       /* Erase screen and
                                    display the molecule */


/* Turn the molecule 6 times in 30 degree increments */


DO I = 1 TO 6;
    TURN 30. AROUND X;
    TYPE NOCR I*30.,' DEGREES'; /* Label each turn */
    IF %YESANSWER('
    COPY') THEN PRINT;           /* Option to copy */
END;                             /* End of DO Loop */

END;                             /* Closing line of MOLTURN */
```

Figure 4.1

Comments can and should be used in the following ways: at the beginning of the procedure to explain the purpose, identify any special conditions or exceptions, and in the body of the procedure to indicate briefly the function of any code which may be slightly obscure.

2. Opening and Closing Lines

The only formal requirement of a procedure definition is that it have the "PROCEDURE" statement in the opening line and a matching "END" statement at the end of the definition. Thus, the example below is a legal, albeit somewhat useless, procedure:

```
PROCEDURE;
END;
```

All statements within a procedure are terminated with a semicolon (including the opening and closing lines).

Note that separating lines of code by carriage-return line-feeds is <u>not</u> required. A definition could in fact be written entirely on one line; however, it is much easier to read later if you use one or more lines per statement.

The opening line of a procedure definition can consist of just the word "PROCEDURE" followed by a semicolon, but there is an optional prefix and an optional suffix which may appear with the word "PROCEDURE". These may be formally expressed as:

```
[procedure-name:] PROCEDURE [ (arguments) ];
```

The square brackets indicate the optional terms.

3. Procedure Names

When you use the MAKE PROCEDURE command, the procedure being made is given a name by which it can later be compiled, edited, and called. It is also possible to give a procedure an "internal" name, i.e., a name which is a part of the definition itself.

If you preface the word "PROCEDURE" in the opening line of the definition with a legal PROPHET variable name followed by a colon, you have given the procedure an "internal" name which, after compilation, it will retain regardless of the name you gave it in the MAKE PROCEDURE command.

Thus if you say in PROPHET

```
MAKE PROCEDURE P
```

and write the first line of the procedure definition as follows:

```
PP: PROCEDURE;
```

when this procedure is compiled, it can be called by <u>either</u> the name "P" or the name "PP".

It is not necessary for a procedure to have an internal name. (The only exception to this is for internal procedures, described below.)

The name you use when you give the MAKE PROCEDURE or the RAC command is usually sufficient for referring to the procedure. Moreover, giving a procedure an internal name can lead to confusion, so it is generally recommended that you <u>not</u> give a procedure an internal name. (Simply for ease of referring to them, all examples of procedures in this manual have internal names.)

4. Arguments of a Procedure

In the formal specification of a definition's opening line:

```
[procedure-name:] PROCEDURE [ (arguments) ];
```

the optional "arguments" suffix implies an important aspect of procedures, viz., the passing of data into and out of procedures.

Within a definition, arguments are simply "dummy" variables which have no real value until the procedure is called. At that time, the "real" variables or expression values given in the CALL command are substituted for the dummy variables within the definition.

Specifying Arguments

If arguments are to be used in a procedure, they *must* be formally specified in the opening line of the definition, immediately following the word "PROCEDURE". The arguments are enclosed within parentheses and, if there is more than one, they should be separated by commas.

Two examples of specifying arguments are:

```
PROCEDURE(TAB);

PROCEDURE(A,B,C,D);
```

The concept of argument passing is a powerful feature of procedures. Using arguments allows you to generalize a set of operations which can be applied to many different variables of the same data type.

For example, if you have several molecules in your workarea and you want to turn them around the X-axis in 30 degree increments, you could use the procedure shown in Figure 4.1 to do that operation on any of your molecules. Thus if you had a molecule named "STEROID", you could say

```
CALL MOLTURN(STEROID)<GO>
```

When the procedure "MOLTURN" is executed, the variable named in the call command -- "STEROID" -- is passed to the procedure. All of the actions performed on the procedure's "dummy" variable -- "MYMOL" -- are performed on the "real" molecule named in the call command. It is as if an assignment statement had been given saying that "MYMOL = STEROID". Execution of the procedure would cause "STEROID" to be displayed on your terminal and turned in 30 degree increments around the X-axis.

Note that all of these operations could be done "by hand" in PROPHET; i.e., you could display a molecule, turn it 6 times and take a hard copy. But if you want to do the same thing for many molecules, the procedure will save you from typing the commands over and over again.

Declaring Arguments

As in the procedure shown in Figure 4.1, all arguments should be declared to be variables of the correct data type. The details of declaring variables are given immediately below.

5. Declarations in Procedures

All variables used in a procedure should be declared. Procedure variables are declared <u>immediately</u> following the opening line of the procedure. No statements (except the RETURNS statement discussed in Section IV.6 below) can intervene between the opening line and the declarations. Declarations serve two purposes: 1) they establish the data-type of the variables named, and 2) they define the scope over which the variables are known. These two points are discussed immediately below.

Declarations are accomplished by means of the DECLARE statement:

```
DECLARE variable-name data-type;
```

If there is more than one variable of the same data type, they can be grouped within parentheses and separated by commas:

```
DECLARE (A,B,C,D) FLOAT;
```

Establishing the Data-Type

The effect of a declaration is to establish the variable-names as objects possessing the attributes of their data-type. No value is assigned to the variables by a declaration. Thus the effect of the example above is to establish "A", "B", "C", and "D" as empty FLOAT variables. For complex data types, the power of the declaration is a bit clearer. If we say in a procedure

```
DECLARE ARA(*,*) FLOAT;
```

the compiler knows that "ARA" is a two-dimensional FLOAT array. This enables PROPHET to interpret correctly later statements which use array subscripts. Similarly, if a variable is declared to be of type GRAPH, you can then write statements which refer to the subfields of a GRAPH. If the GRAPH or array were not declared, such statements would not compile.

For such simple data types as FLOAT or FIXED, you might use undeclared variables within procedures, but the practice is not a good one; as we shall see below in Section V.A.6, it can often lead to inefficient execution as well as to other more serious side effects.

The Scope of Variables

When a variable is declared within a procedure, it has no effect on variables outside the procedure--*even on variables of the same name*. The "scope" of a variable declared within a procedure definition is only within the procedure. The only way to "move" the contents of a procedure variable outside the procedure is through the mechanisms provided by argument passing, the RETURN statement, or by I/O statements and procedures. Note that even when one of these mechanisms is used, it is the *contents* of the procedure variable which is "moved", *not* the variable-name.

Global Variables

When a procedure is being compiled and a variable name is encountered which has *not* been declared, the compiler looks to see if there is a variable of the same name present in your PROPHET workarea. If there is, it will use that top-level variable within the procedure. Such a variable is said to be "global" to the procedure, i.e. it is known outside the procedure.

Implicit Declarations

If the compiler fails to find a variable either within the procedure or in top-level PROPHET, it "implicitly" declares the variable to be of the pseudo data type ANY. When this occurs, PROPHET will type a message identifying the variable being implicitly declared. So long as the undeclared variables are of simple data types, the procedure will probably compile and execute, but it should be stressed again that this is not a good programming practice, because such variables are global, i.e. their values are known outside the procedure.

6. Functions

In Section III above, several system FUNCTIONs were described. A FUNCTION is a special kind of procedure that returns a value. As such, it can be used directly in an assignment statement. For example, if F is defined as a function that returns a FLOAT value, it is legal to say

```
Y = F(X)
Z = Y + F(X)
```

where "Y" and "Z" are FLOAT variables.

Requirements for a Function

The definition of a function differs from a procedure in two ways:

(a) The line immediately following the opening line of the definition must be of the following form

```
RETURNS data-type;
```

where "data-type" is the data type the function will return. In our example above, where F was defined as a Function returning a FLOAT value, the first few lines of the definition might be:

```
F:  PROCEDURE(X);
RETURNS FLOAT;
DECLARE  (X,ANS)  FLOAT;
  . . . .
```

(b) The second requirement of a function is that when the procedure has arrived at the "answer" that is to be returned as the value of the function, you must have the statement

```
RETURN return-value;
```

```
/* A function to calculate the tangent of an
   angle measured in radians */

TANGENT: PROCEDURE(RADS);          /* Angle is in radians   */
    RETURNS FLOAT;                 /* Procedure is function */

DECLARE (RADS,T) FLOAT;

T=SIN(RADS)/COS(RADS);             /* Compute tangent
RETURN T;                             and return it */

END;                               /* of TANGENT */
```

Figure 4.2

where "return-value" may be the local name of a variable whose type is the same as that used in the "RETURNS data-type" statement, and contains the answer you want returned. "Return-value" can also be an expression which, when evaluated, is the value you want returned.

There can be more than one RETURN statement within a function, but only one will be executed for any given call to the function.

An example of a function is shown in Figure 4.2 on the facing page. This function "TANGENT" uses the system functions SIN and COS to compute the tangent of an angle measured in radians. Once it is compiled, it can be used directly in mathematical expressions, or indeed in any context where a FLOAT variable or constant would be allowed.

7. Multiple Entries in Procedures

Usually execution of a procedure starts at the first line and continues from the top down. Occasionally, it is useful to have multiple entries into a procedure, i.e., more than one way to call it. The mechanism for doing this is the ENTRY. It is analogous to the opening line of a procedure, with the word "PROCEDURE" replaced by the word "ENTRY".

```
entry-name: ENTRY [(arguments)] ;
```

The "entry-name" is required, but the "argument" suffix is optional. An ENTRY can also be made to act as a function by immediately following it with a RETURNS statement.

An ENTRY need not have the same list of arguments as the procedure containing it. Using multiple entries

```
/* A function to calculate the tangent of an
   angle measured in degrees or radians */

TANGENTD: PROCEDURE(DEGS);      /* Angle is in degrees    */
    RETURNS FLOAT;              /* Procedure is function */

DECLARE (RADS,DEGS,T) FLOAT;

RADS = %PI * (DEGS/180.);       /* Convert degrees to
                                   radians */

TANGENT: ENTRY(RADS);           /* Angle is in radians */
    RETURNS FLOAT;              /* Entry is function    */

T=SIN(RADS)/COS(RADS);          /* Compute tangent
RETURN T;                          and return it */

END;                            /* of TANGENTD */
```

Figure 4.3

in procedures is recommended only when conceptually and actually the procedure and the entry share a substantial amount of common code. For example, a procedure may require no arguments and ask the user at run-time to specify the identity of the data to be used. If you also want the ability to call the procedure with a list of arguments and not have to answer questions, then an ENTRY is probably the right solution.

In the example shown in Figure 4.3, we expand on the "TANGENT" function described in Figure 4.2. Where the earlier function calculated the tangent of an angle measured in radians, we can add an entry which makes it capable of calculating the tangent of a function measured either in radians or degrees. Note that the code has been slightly restructured to minimize the number of statements required.

If the function in Figure 4.3 is called using the name in the opening line, the degrees used in the argument will be converted to radians. The entry "TANGENT" will be skipped and the tangent will be computed as before. If the entry "TANGENT" is called, execution will immediately pass to the line that calculates the tangent, dividing the sine by the cosine.

8. Internal Procedures

Like any other data type, procedures can be declared and used within procedures. When a procedure is wholly contained within another procedure, the contained procedure is called an "internal procedure".

An internal procedure must be declared within the containing procedure and the opening line of the internal procedure must have an internal name which is identical to that used when it is declared.

It is important to emphasize that declarations within an internal procedure have scope only within the internal procedure. For example, if SUB1 and SUB2 were both internal procedures within the procedure MASTER, it is possible to have variables of the same name declared in all three procedures. (Such a situation is not recommended, however, except for trivial cases of, say, DO Loop counters.)

If a variable is not declared within an internal procedure, the compiler will first look in the parent procedure's variables to see if a variable of the same name is declared there. If it finds the variable in the parent procedure, it will use that variable within the internal procedure. Such a variable declared at the higher level but used within the internal procedure is said to be "global" to the internal procedure. While using such global variables within internal procedures is permissible, it is one of the most common causes of runtime errors, so the best advice is to avoid using globals and to pass all needed information explicitly as arguments to the internal procedure.

Internal procedures are mainly useful when you are writing a fairly long and complicated procedure. It is often helpful to break the procedure up into several pieces each of which performs a specific function and may be an internal procedure. This is especially useful if the same operation must be repeated in several different contexts within the main procedure. Rather than repeating similar code several times you can call an internal procedure with different arguments.

B. Notes of a Procedure

The NOTES of a procedure is a TEXT which if not EMPTY, is typed out whenever the procedure is restored from a saved table. Procedure NOTES are particularly useful for giving short instructions about using the procedure, or for announcing changes that have been made. To give a procedure NOTES, the command is

```
SET NOTES OF procedure-name TO notes-text
```

The NOTES may serve as an adjunct to the comments of a procedure.

C. Compiling Procedures

Before a procedure can be used in PROPHET, the PL/PROPHET statements contained in the DEFINITION subfield of the procedure must be compiled; i.e., they must be "translated" to the machine code understood by the computer. When you exit from the PROPHET Editor after making or editing a procedure, you are asked "COMPILE?" If you answer with "YES" the compilation begins. If the procedure successfully compiles, the word "DONE" will be typed out. Because compilation can be a time-consuming process, PROPHET saves the compiled code along with the definition of a procedure whenever you use the SAVE command. Then when you use the RAC (RESTORE and COMPILE) command, the procedure does not need to be compiled again.

Recompiling Procedures

There are a few times, however, when the code of saved procedures must be re-compiled. Whenever a new version of PROPHET is installed, the location of many system variables changes. Inasmuch as the compiled code of procedures refers to these symbols by location, not by name, the code, if it could be executed, would do exceedingly strange things. Consequently whenever PROPHET detects that the compiled code of a procedure is from a different version of PROPHET, it will automatically initiate a recompilation when you use the RAC command.

To save time, there is a public procedure which is designed to recompile all procedures in a saved table. A description of "COMPSAVED" can be found in the Public Procedures Notebook.

Note, too, that compilation can always be made to occur by using the %FORCECOMPILE command.

D. Using Procedures

Section IX of the Users Manual contains basic information about using PL/PROPHET procedures. The following are some more advanced pointers on using procedures.

Using Arguments in CALLs

Once a procedure is compiled, arguments are specified in a call to a procedure by enclosing them in parentheses and, if there are more than one, separating them by commas. Arguments named in a call to a procedure can be variables or constants of the correct data type, or expressions which evaluate to the correct data type. Thus if a procedure has one argument of data type FLOAT, it must be formally declared within the procedure definition to be a FLOAT variable. Once the procedure is compiled, you can call it with the name of a FLOAT variable, e.g., %PI, or a FLOAT constant -- e.g., 4.3.

When a procedure is called, the correct number of arguments _must_ be specified or an error message will be given: "WRONG NUMBER OF ARGUMENTS".

"Empty" Arguments

For a few procedures, it may be safe to leave out the name or value of an argument, but even in those cases, the correct _number_ of arguments must still be specified. For example, the calling sequence of the system procedure "%C2A" is:

```
%C2A(table-name,col-id,first-row,last-row,array-name).
```

If you want "first-row" to be 1, and "last-row" to be the last row of the table, you can omit those argument variables or constants from the call:

```
CALL %C2A(MYTAB,1,,,ARA)
```

but note that the correct <u>number</u> of arguments (as indicated by the commas) must still be given. The corresponding variables within the procedure will be EMPTY. In this case, the procedure %C2A checks these variables and if they are EMPTY sets them to default values.

V. Writing and Debugging PL/PROPHET Procedures

A. Programming Guidelines

1. Designing Procedures

In this section, a great many programming guidelines are suggested and discussed. Of course, the first priority in writing effective procedures is to come to grips with the problem or problems the procedure is intended to solve. This seems self-evident, but writing programs before the problem is thoroughly thought through is a frequent cause of wasted effort. Some general suggestions to prevent this are:

a. Carefully consider the kinds of data that will be used and produced by the procedure. If there is an existing data base, make sure that the procedure is designed to utilize the structure of that data base most effectively. If a procedure is going to create a new data base, spend time making sure that the organization of that data base is convenient for the persons who will be using it, and is reasonable in the demands it places on system resources -- particularly on disk space.

b. It is generally a good practice to organize the procedure before you go to the PROPHET terminal and use the MAKE PROCEDURE command. The best approach is to break down the overall problem into its constituent sub-problems. The really difficult parts of the problem can then be identified and separated from the less complex areas. Isolating these more complex sections and developing the code for them first is usually a good idea; often solutions to the more difficult sections of the procedure will necessitate hitherto unforeseen complications in the "simpler" sections.

c. When your procedure is computational (e.g., a statistical program), it is a good practice to have a "standard" problem for which the answers are known and against which the results of your new procedure can be compared.

2. Writing Readable Code

PL/PROPHET has very few hard and fast rules about the format of a procedure's definition. Except for the necessity of the words PROCEDURE and its matching END, and the need to end each statement with a semi-colon, there are virtually no other requirements for writing code. Lines can be broken at any spot, indented or not indented, commented or not commented.

There are, however, several reasons why it pays to adopt some personal rules of formatting and commenting code: maintaining and revising code is much easier with comments and a coherent formatting scheme -- when your code is read by another person, a lack of comments and consistent format make obscure even the most brilliant code.

Here are a few ways of making code more readable:

a. Use terse comments to indicate why a particular operation is being done. "Why" is more important than "what" since what is happening is usually clear from the code.

b. Use a consistent indentation scheme to visually indicate the scope of DO . . . END Blocks and DO Loops. A common scheme is to indent 4 spaces from the left for every DO and stop indenting the 4 spaces when the closing END of the DO is reached. When this practice is followed, it is almost always easy to find cases of an unmatched or an extra END.

c. It is also useful to indent complicated IF . . . THEN and ELSE statements so that the relationship of the ELSE to the matching IF is clear.

d. Be generous in the use of carriage returns and spaces to visually set off sections of a procedure. Avoid using lines that are longer than the width of the screen.

e. Do not have more than one statement per line.

f. Mnemonic variable names are highly recommended. It is easy to forget exactly what "A", "B", or "C" are when you get to the tenth page of a procedure. Such names as "TOTSUM", "NCOLS", "NROWS", and "GRAF" are more easily recognized and remembered. At the very least, use comments after the declaration of variables to define your cast of characters.

3. Writing Structured Code

As discussed above in Section IV, PL/PROPHET is a block structured language. This feature of the language should be used when you are writing your procedures. At a minimum, structured code means using DO . . . END Blocks and DO Loops in place of GO TOs and Labels; the resulting code will be much more comprehensible and easier to maintain and revise.

Another aspect of structured code is implied in Section V.A.1 above: breaking the overall problem being addressed by the procedure into its constituent parts and developing solutions for those parts. This approach lends itself to the use of internal procedures within the larger procedure, each of which is designed to "solve" a particular aspect of the problem. A great advantage of this is that these internal procedures can initially be written separately and tested in isolation from the other parts of the parent procedure.

While internal procedures can appear virtually anywhere within the definition of the parent procedure, for readablity it is a good idea to begin the internal procedure definitions immediately following the declarations for the parent procedure or at the very end of the parent procedure.

best copy available

4. Interactive and Quiet Procedures

Another design question to keep in mind is whether the procedure will be used in an interactive way -- with frequent questions to and responses by the person invoking the procedure -- or "quietly" with just a CALL command. It is often useful to write the procedure so that it has the flexibility to function either way -- if the user wants to be prompted for the data to be input, he can use one entry; if he wants to specify all input variables in the calling sequence, he can be provided with a "quiet" entry. An example of this is the public procedure "STAT", which is interactive if called by its name "STAT", but which is quiet if the entry "STATQT" is used. An attempt has been made to provide several system and public procedures with such "quiet" entries: normally the quiet entries have the suffix "QT" added to the procedure's "interactive" name.

5. Size and Space Considerations

It is important to pay attention to questions of space and size at an early stage of procedure design. While PROPHET can handle a considerable amount of data and number of variables within a procedure, there is a limit to both the number and the size of variables that can be worked with at one time. Procedures which make several large GRAPHs or MOLECULEs will almost certainly run into difficulties and receive a "SHORT OF SPACE" message -- from which there is really no escape.

If care is taken in writing the procedure, these size and space problems can usually be overcome. One useful technique is to use the CLEAR statement once you have finished with variables--particularly variables such as GRAPHs, MOLECULEs, and arrays which are temporary variables and as such exist solely in memory. (Permanent tables are stored on disk and consequently only small portions of a table will be in memory at a given moment.)

Some further guidelines on size and space considerations are given below as adjuncts of efficiency concerns.

6. Using Global Variables

It is generally recommended that you not write procedures which depend on global variables -- i.e., on variables which are known outside the procedure. The reason for this is simple enough: if a procedure depends upon some global variable to be present in your workarea, it will almost certainly behave badly if that global is absent (e.g., if you have forgotten to restore some saved variable, or if a table needed by the procedure has been archived). Also, the code can be hard to understand since the globals are never explicitly defined.

There are a few times when such use of global variables can serve a useful purpose. Perhaps you want your procedure to call a public procedure. There is no need to copy the definition of the public procedure; if the use of such a global procedure is handled carefully, no problems should be presented.

Some suggested guidelines for using global variables are:

a. Declare all global variables which a procedure is going to use. This is accomplished by ending the normal DECLARE statement with an exclamation point instead of the usual semi-colon. Thus if your procedure "MYPROC" wants to use the public procedure "STAT", the beginning lines of your procedure would be:

```
MYPROC: PROCEDURE(args);
DECLARE STAT PROCEDURE! /* Global Public Procedure */
DECLARE (args) data type;  /* Local declarations */
```

By declaring all globals, you assure that the compiled code will know the correct data type of the global variables and will not fail to compile because of a type mismatch. Also, you should document with comments what the globals are and how they are used.

b. When a procedure uses globals, have the code check to see that the global variable is present in your workarea and in its expected state when <u>your</u> procedure is called. Thus in the example above, it would be a good idea <u>not</u> to rely on the user having to remember that the public procedure "STAT" needs to be restored and compiled before he calls "MYPROC". This can be done by "MYPROC" itself:

```
IF EMPTY(NOEVAL STAT)
    THEN RAC PUBLIC STAT;
```

If this line is included in "MYPROC" before the line which calls "STAT" you can be assured that your procedure will run. (The NOEVAL statement is required when you want to use the name of a procedure without calling it; it is described in Section VI below.)

B. Common Compile-Time Problems

Problems with procedures tend to fall in two categories: those that arise when the procedure is being compiled, and those which occur when the procedure is being executed. The latter are discussed in the following section; here we describe some of the most common compile-time errors -- how they are reported, and what diagnostic tips the compiler gives you.

1. Reporting of Compile-Time Errors

When the compiler detects an error in the code it is compiling, it types out an error message followed by the section of the code which caused the error. This section of the code is labelled "LAST LINE PROCESSED". A typical example of this reporting is:

```
[COMPILING...89. MISSING ) OR , OR ( IN DECLARATION
LAST LINE PROCESSED:"E,
    (BIGTEMPO,TEMPO) FLOAT,
    (INDARA) FIXED ARRAY,
    (I,J,K,L,M,          ***
"
IN SYS. PROC.  371
...FAILED TO COMPILE]
```

When interpreting the "Last Line Processed" message, remember that the line triggering the error message is the *last full* line shown. (In the example above, it is flagged with "****".) It is *this* line which has caused the error to be detected, but as will be described below, the line itself may or may not actually *contain* the error. Usually, the error message together with the last line processed is sufficient to pinpoint the error exactly. There are, however, some cases which can be more difficult to diagnose and these are discussed in the subsections below.

2. Unmatched Quotes

One of the most common and perplexing compile-time problems is caused by leaving out the closing quotation mark when using TEXT constants. This can be difficult to detect, since all characters after the opening quotation mark are interpreted as part of the TEXT until a closing quotation mark is found. If the closing quote is missing from its intended spot, all succeeding characters are regarded as part of the TEXT until the next quotation mark is found. Since that quote is probably the opening quotation mark of another TEXT, an error will usually result because the second TEXT is now treated as a statement.

A good rule of thumb in detecting this problem is that if the "Last Line Processed" contains a quotation mark, there is a very good possibility that this quotation mark is being matched with a previous quote which has not been closed at the proper spot. Note that there may be nothing actually wrong with the "Last Line Processed" itself; but if it contains a quotation mark, you ought to look at the lines preceeding it for an unclosed quote.

3. Unmatched or Extra ENDs

When you receive the error message "ILLEGAL STATEMENT IN BLOCK" you have probably failed to close a DO with an END. The best way to avoid this is to scrupulously adhere to a consistent indentation scheme for DO . . . END Blocks and DO Loops. This is usually sufficient to detect unmatched DOs visually.

A related problem is having an extra END in the code. This can cause the compiler to interpret the extra END as the closing END of the procedure and conclude the compilation prematurely! Again the solution is to use indentation to prevent such a problem from arising.

4. Missing Semi-colons

Another frequent cause of compile-time errors is omitting the concluding semi-colon from a line of code. The compiler then tries to combine the unterminated line with the next line of code. Since the resulting code is unpredictable, the error messages generated can be quite puzzling. The best way to detect this problem is to carefully examine the "Last Line Processed" and to make sure that the last line and the preceding line are correctly terminated by semi-colons.

5. Unmatched Parentheses

The PL/PROPHET compiler tries to detect cases where the wrong number of parentheses occur and attempts to inform you that it has found a mismatched set of open and closing parentheses. Typically the "Last Line Processed" will contain the mismatch, but occasionally the open and closing parentheses may be spread across a couple of lines.

6. Collisions with Abbreviations

As stated earlier in Section IV, the names of local variables in a procedure will not collide with top-level variables having the same name. The only exception to this occurs when you have used the ABBREVIATE command to set up an abbreviation. If this is done, and a variable within the procedure is given the same name as the abbreviation, the compiler tries to expand that name to the value of the abbreviation. Since this usually makes no sense at all within the context of the procedure, an error will be detected. Unfortunately, the form of the error message will be wholly dependent on what was being abbreviated; i.e., the error message will be identical to the one which the value of the abbreviation would trigger if it appeared in the procedure. Thus if you say at top level PROPHET "ABBREVIATE 'MOVE TO ROW' AS M", and you then write a procedure where the statement "DECLARE M FLOAT", the compiler will act as if you had said "DECLARE MOVE TO ROW FLOAT;". If you routinely use abbreviations, the best advice is to be careful with the variable names you use in procedures or with the names you give your abbreviations.

7. Unclosed Comments

Another compile-time error that may be difficult to diagnose can be caused by an unclosed comment, i.e. a missing */.

The compiler will often catch this error and report it, but there can be cases where the procedure will actually compile and then run in some quite startling ways.

C. Common Run-Time Problems

In addition to errors that are reported when you try to compile a procedure, a successfully compiled procedure may produce errors when you try to execute it; these are called run-time errors.

1. Reporting of Run-Time Errors

When a run-time error occurs, PROPHET will usually type out an error message which attempts to explain the problem that was encountered. Often this error message will be sufficient to identify the problem in the procedure and fix it. At other times, the error message may not seem to be of much assistance in finding the problem, but there are techniques which can assist you in interpreting these error messages.

2. Undefined Variable References

Perhaps the most common run-time error message is 128. "UNDEFINED VARIABLE REFERENCE". What this simply means is that an EMPTY variable has been encountered and some operation has been attempted (e.g., an arithmetic calculation, or concatenation) which cannot be done with an EMPTY value.

Many undefined variable references are caused by the failure to initialize the procedure's variables. It must always be remembered that PL/PROPHET variables do *not* have a default initial value. Until they are assigned values, they are EMPTY. Also, if data are typed in at run-time, an EMPTY value may have been entered.

The way to prevent most occurrences of undefined variable reference is to be generous in testing with the EMPTY and NOTEMPTY functions in your code. This will usually catch unexpected empties, allow you to take corrective action, or if continued execution is impossible, you can at least report what variable was empty.

3. Some Common Arithmetic Errors

Perhaps the most common arithmetic error is caused by the attempt to divide a number by zero. Again prevention is the best cure. If there is any possiblity that the divisor of an arithmetic expression could be zero, rewrite the code to explicitly check for the possibility.

The similar error caused by the attempt to take the logarithm of a negative number can be handled in the same way.

Another common arithmetic error is caused by attempting to perform arithmetic operations on numbers which are larger (or smaller) than PROPHET can handle. When this occurs the error messages "FLOATING UNDERFLOW" or "FLOATING OVERFLOW" will be given. The minimum and maximum values that PROPHET can deal with (%MAXPOSFLOAT, %MAXNEGFLOAT, %MINPOSFLOAT, %MINNEGFLOAT, %MAXFIXED, %MINFIXED) are given in Section III. If there is any chance that your code can produce numbers which approach these limits, you should include statements in the code which check for any proximity to the maximum and mininim limits.

4. Roundoff Errors

The PDP-10 system on which PROPHET runs uses a binary representation for numbers. This will occasionally result in some slight discrepencies because of roundoff errors occuring when converting decimal to binary, and vice-versa. These roundoff errors normally present no difficulties if you remember these guidelines:

a. Be very cautious when comparing two FLOAT variables if one or both is the result of a previous calculation within a procedure. For example, due to roundoff error, the actual value of X may be 5.99999999. If you TYPE the variable it will be output as 6., but the test X < 6. will be TRUE. Useful techiques to avoid this problem are to use the %ROUNDOFF or %SIGFIG functions to resolve system-caused roundoff problems, or to use a "tolerance factor" when making comparisons of FLOAT numbers, e.g.

```
IF ABS(X-6.) <= 1.E-10  THEN . . .
```

b. A more serious problem arises when an exit from a DO Loop depends on a FLOAT variable falling beneath (or above) some other FLOAT value. Roundoff errors can easily cause the loop to be terminated prematurely, or, in some cases, to become an infinite loop. An example of this might be:

```
DO I = 1 TO Z WHILE X < 1.;
    X = X + I*Y;
END;
```

Such a loop can terminate one iteration too soon or one iteration too late because roundoff error can cause X to be just slightly greater or slightly less than 1.

5. Side Effects on Arguments

Another common run-time problem is caused by unintentionally changing the value of an variable used in a procedure's arguments. This can best be shown by the example below:

```
A = 5.28;
CALL MYSUBPROC(A,B,C);
SUMM = A + C;
```

If you are assuming that "A" in line 1 and "A" in line 3 have the same value, you must be careful in the coding of the subprocedure "MYSUBPROC" <u>not</u> to reassign values to <u>its</u> first argument variable. Thus if the code for "MYSUBPROC" were:

```
MYSUBPROC: PROCEDURE(A,B,C);
DECLARE (A,B,C) FLOAT;
A = (A+B)/(A-B);
C = A + 1./B**2;
END;
```

line 3 will have changed the value of "A" to (A + B)/(A-B) -- <u>both within</u> "MYSUBPROC" <u>and in</u> the parent procedure.

The best way to avoid this problem is to be <u>extremely</u> careful about changing the value of <u>any</u> variable used as an argument. Only when you are very sure that you really want the value of an argument variable to be different at the end of a procedure's execution should you change its value. Comments at the beginning of the procedure should state all side effects of the procedure on its arguments or on any globals referenced. For complex variables you can avoid such problems by using the COPY function to set up copies that you can alter. In our example above, the problem could be avoided by using a local variable within "MYSUBPROC" which could be assigned the result of the calculation in line 3.

D. Efficiency Considerations

This section deals with an area of program design that sometimes gets too little attention -- efficiency. By efficiency we mean writing code that accomplishes its tasks using the minimum amount of run-time and real-time and other system resources. Note, however, that code which executes faster usually is somewhat extravagant in its use of space, while code which tries to use the least space usually takes somewhat more time.

If your procedures consist of just a few lines of code, you may not have much concern about the efficiency of its execution. But if those few lines consist of a DO Loop, you should probably give some thought to how efficient the procedure is. Several techniques are suggested below to assist you in producing efficient code.

1. Pre-allocation of Arrays

Perhaps the most frequent cause of inefficient procedures is the failure to pre-allocate arrays. When an array is declared of variable length, i.e., with an asterisk, it has an initial length of 0. When you later begin to assign values to elements of the array, space must be found for the array. Assigning values in the order 1, 2, 3, . . . N means that for every element, PROPHET must find a bigger space for the array, and the entire array up to that point is copied into the new space. If the array is large, this can be highly costly in run-time and can easily be prevented.

The "ALLOCATE" statement can be used to establish the maximum length of an array before any data is assigned to the array elements. The form of the ALLOCATE statement is:

```
ALLOCATE array-name(length1[,length2, . . ])
```

where "array-name" is the array being pre-allocated and "length1", "length2" etc., are the maximum lengths of each dimension of the array. Thus if you know that the maximum length of a one-dimensional array is 500, you can say

```
ALLOCATE DATARA(500)
```

A more useful version of ALLOCATE employs a FIXED variable with differing run-time values:

```
ALLOCATE DATARA(NROWS)
```

For a two-dimensional array, the form might be

```
ALLOCATE MATR(NROWS,NCOLS)
```

where "NROWS" and "NCOLS" are FIXED variables containing the maximum length of each dimension. Note that the allocation is done at run-time. This contrasts with fixed length arrays whose space allocation is done at compile-time. Allocation should usually be done just before the array is needed, or you will be carrying around empty space you don't yet need.

Note that <u>no</u> data is assigned to the array when the ALLOCATE statement is used -- all you have done is to let PROPHET know how large the array is going to be.

2. Redundant Code

Another common cause of inefficiency is the use of redundant calculations and redundant tests, particularly in DO Loops. An example is:

```
DO I = 1 TO 1000;
    DAT1(I) = A * B(I) - B(I)**2/3.;
    DAT2(I) = A * B(I) + B(I)**2;
END;
```

Note the following inefficiencies in this example: (a) the value stored in the Ith element of the array B is "fetched" 4 times, (b) the value of B(I)**2 is computed twice, and (c) the value of A * B(I) is computed twice. Since these calculations are going to be performed 1000 times, you are going to perform some 5000 extra calculations or assignments. Such redundencies can be prevented by doing the calculations <u>once</u> and introducing some FLOAT variables that can be used as "pointers" to hold the values of these calculations. For example,

```
DO I = 1 TO 1000;
    P1 = B(I);
    P2 = P1**2;
    P3 = P1 * A;
    DAT1(I) = P3 - P2/3.;
    DAT2(I) = P3 + P2;
END;
```

Now no calculation is done more than is absolutely necessary and no array element is referenced more than once for each iteration in the loop. The use of such "pointer" variables makes the procedure's code slightly larger, but it will greatly increase the execution speed.

Some good guidelines for this problem are:

a. Whenever you reference the same element of an array more than once, assign the value of the array element into another variable, especially when you are not changing the value of the array element itself.

b. If the same calculation, or call to a function (such as SQRT, SIN, etc.) is done more than once, use a variable to hold the value.

Another example of code that can cause inefficient execution is shown in this example:

```
NC = %LASTCOL(TAB);
NR = %LASTROW(TAB);
TOTSUM = 0.;

DO I = 1 TO NC;
    DO J = 1 TO NR;
    /* Add cell if it's a number */
        IF (%GETCTYPE(TAB,I) = 'NUMBER')
           THEN TOTSUM = TOTSUM +
                            %GETCELL(TAB,J,I);
    END;
END;
```

The problem with this code is that the %GETCTYPE function is in the inner loop of the two DO Loops. This means that it will be called not just once for each column, but for each row as well! If the table in the procedure had 100 rows and 20 columns, then %GETCTYPE would be called 2000 times instead of 20.

The point to be made is never do calculations in an inner DO Loop that can be done in the outer loop. If you find yourself in this situation, the first thing to do is to determine if the problem can be solved trivially (as in this example), or, if necessary, consider switching the order of the loops. An efficient re-coding of this example is:

```
DO I = 1 TO NC; /* Is column of right type? */
    IF (%GETCTYPE(TAB,I) = 'NUMBER')
         THEN  DO  J = 1 TO NR;
            TOTSUM = TOTSUM +   /* Add cell */
                 %GETCELL(TAB.J,I);
        END;
END;
```

Note that not only do you reduce the number of calls to %GETCTYPE, but the inner loop counter never needs to be initialized and incremented except for those columns that pass the test.

3. Nested IFs vs ANDs

While the use of the boolean operator AND can be very helpful, there are times when it is best to eschew its use for a somewhat less elegant, but more efficient series of "nested IFs". An example is:

```
DO I = 1 TO 100;
    A = ARA(I);
    IF (A < 100.) AND (A > 0.)  /* A in range? */
       THEN Y = Y + A;
END;
```

The problem with this code is that both sides of the AND expression will always be evaluated. Depending on which test in the example is <u>most</u> likely to be FALSE, the code can be speeded up by testing for it first. Let us assume that there are only a few numbers greater than 100, but many which are less than 0. If that is so, then the code would be much more efficient if written as:

```
DO I = 1 TO 100;
    A = ARA(I);
    IF (A > 0.)          /* A positive? */
       THEN IF (A < 100.)   /* A small enough? */
           THEN Y = Y + A;
END;
```

In the first example, a total of 200 comparisons would be made; 2 for each iteration. In the second, all numbers would be tested for the condition (A < 0.), but only those that succeeded in meeting that criterion would be tested for (A > 100.) This use of nested IFs is most important within DO Loops. Outside of DO Loops the use of AND is probably preferred because of its greater clarity.

4. Timing Execution of Procedures

One way to detect inefficient code is to include timing calculations in the procedure definition itself. The RUNTIME function described in Section III can be used to give you accurate information on the amount of run-time the procedure uses and it can also be used to time smaller sections of the program. A common technique is to use a FLOAT variable which is given the amount of run-time at the beginning of the procedure, and then subtract that variable from the amount of run-time at the end of the procedure. An example might be:

```
MYPROC: PROCEDURE;
DECLARE (TEMPO) FLOAT;
        .
        .
        .
TEMPO = RUNTIME; /* Store value of
                    run-time at beginning */
        .
        .
        .
TYPE NOCR 'CPU SECS. USED = ',RUNTIME-TEMPO;
END;        /* of MYPROC */
```

If you suspect that your procedure is using more run-time than seems reasonable, it is advisable to determine what sections are using the largest amounts of run-time and then to see how they can be made more efficient.

5. Using CLEAR and PARE

In general, it is a good practice to carry around the minimum of data that is absolutely required for the task at hand. This is particularly true for such large data objects as arrays, graphs, and molecules. Carrying around lots of extra objects will tend to slow down the execution of the procedure, as well as response for everyone using PROPHET at the same time.

Thus if the last thing your procedure does is to display a graph, it is recommended that you put off making that graph until it is really needed. Even if it is possible to make the graph in the first 15 lines of the procedure, there is no need to take up the core space used by the graph until you want to display it.

Similarily if you make a graph in the middle of your procedure and do not use it again, it is a good practice to use the CLEAR statement immediately after you have finished with it.

Paring Tables

PROPHET allows you to have a total of 6 tables "open" at a time -- i.e., a procedure can be referencing no more than 6 tables at a given moment. If the procedure exceeds this limit, you are told "TOO MANY OPEN TABLES, CLOSING ONE", and one of the tables is arbitrarily closed. Since you probably have a better idea about which tables you still need, you can prevent this by using the PARE statement as soon as you are finished using a table in a procedure. The structure of the command is simply

```
PARE local-table-variable-name
```

All PARE does is recover the in-core space used by the table <u>without</u> affecting the table on the disk. If a PAREd (or a closed) table is referenced again later, it must be found on disk again (so you pay a time penalty). This is a good example of space/time trade-off.

E. Debugging Techniques

A program is said to have a "bug" in it when it produces an error for some set of data or when it executes without error, but does not produce the expected result. In this section we describe a few techniques for finding the causes of bugs.

1. Examining Values of Variables During Execution

If your procedure is producing a wrong result or if some puzzling "Undefined Variable References" persist in causing your procedure to abort, it is often useful to temporarily recode the procedure by sprinkling TYPE or DISPLAY statements liberally throughout the procedure to output intermediate values of any key variables which may be contributing to the problem. A similar approach is often useful if you get "FLOATING OVER/UNDERFLOW" errors, or if you are suspect you are of getting roundoff errors.

2. %TRACEDEPTH

When an error occurs after calling a procedure, you normally get 1 line of error reporting. This is usually sufficient, but if you use internal procedures, or want a more detailed report, it is often useful to have more information typed. This can be done by changing the value of the global variable %TRACEDEPTH. The command is simply

```
%TRACEDEPTH = desired-level
```

where "desired-level" is the FIXED number of lines of errors you want typed out. This command should be typed to PROPHET just before you call the procedure which is giving you the error. Then, when the error is encountered, the desired number of error reporting lines will be typed.

3. Using %REFS

When undeclared variables are encountered as a procedure is being compiled, you are usually told that an implicit declaration is being made. You can also get a detailed list of all global references made by a procedure by using the %REFS statement. Its form is

```
%REFS(NOEVAL procedure-name)
```

This will cause a complete report to be typed out which specifies the name of any global variable (whether declared or not) referenced by the procedure.

4. Using %%LCLS

When an "Undefined Variable Reference" is encountered, you will sometimes be told in the error message that "VARIABLE IS %%LCL(n)". This %%LCL number can be used to pinpoint the actual variable which is causing the trouble. Essentially the %%LCL reference is simply a count of the local variables within a procedure. The rules for counting the %%LCLs are as follows:

a. Counting starts with the arguments, which are not counted again. (This is one good reason to declare all argument variables at the very beginning of the procedure's declarations.)

b. Then proceed counting all variables in the order in which they are declared (excluding any global declarations).

c. Labels should be counted whenever they are *first* referenced.

When your procedure has internal procedures, the counting of %%LCLs is somewhat more complicated. When you arrive at the beginning of an internal procedure, stop counting for the parent procedure. Then begin counting the %%LCLs for the internal procedure at that "Base-number" + 1. Counting for the internal procedure is done exactly as described above. When you are through with the first internal procedure, begin counting the %%LCLs for the second internal procedure (assuming there is one) at the same "Base-number" + 1.

```
MYPROC:PROCEDURE(TAB,GRAF);   /* %%LCL(1) AND (2) */

DECLARE TAB TABLE;            /* Already counted */
DECLARE (I,J,K,L) FIXED;      /* %%LCL(3) - (6) */
DECLARE GRAF GRAPH;           /* Already counted */
DECLARE TEMPO FLOAT;          /* %%LCL(7) */
DECLARE (T,TT) TEXT;          /* %%LCL(8) and (9) */
DECLARE (ANGCALC,ROWVEC)
         PROCEDURE;           /* %%LCL(10) and (11) */

*  *  *  *                    /* Statements of MYPROC */

ANGCALC:PROCEDURE(X,Y);       /* %%LCL(12) and (13) */
RETURNS FLOAT;

DECLARE (X,Y)(3);             /* Already counted */
DECLARE Z FLOAT;              /* %%LCL(14) */
DECLARE (I,J) FIXED;          /* %%LCL(15) and (16) */

*  *  *                       /* Statements of ANGCALC */

END;                          /* of ANGCALC */

*  *  *  *                    /* More statements of
                                 MYPROC */

ROWVEC:PROCEDURE(W,Z);        /* %%LCL(12) and (13) */

DECLARE (W,Z) FLOAT;          /* Already counted */
DECLARE (M,N) FIXED;          /* %%LCL(14) and (15) */

*  *  *                       /* Statements of ROWVEC */

END;                          /* of ROWVEC */

*  *  *  *                    /* More statements of
                                 MYPROC */

END;                          /* of MYPROC */
```

In the example above, %%LCL(5) is K; the two internal procedures are declared within MYPROC. Note that you begin counting at 12 for the first internal procedure, ANGCALC, <u>unless</u> you have referenced a LABEL in the statements indicated by the first group of four asterisks. Similarly, you again begin counting %%LCLs at the same number you used for ANGCALC for the second internal procedure, ROWVEC, unless you have referenced a LABEL for the first time in the statements indicated by the second group of four asterisks. Usually, of course, your internal procedures will all be grouped either at the very beginning or at the very end of the parent procedure.

VI. Advanced

A. The NOEVAL Statement

Whenever you mention
PROPHET, even if it is n
PROPHET tries to i
Sometimes, e.g. in testi
RACed or in using %REFS
don't want to call t
this automatic invocatio
its name with NOEVAL.

```
IF EMPTY(NOEVAL GLO
```

B. The EVAL Function

EVAL is a very po
to perform some opera
otherwise possible wit

The full structure

```
EVAL(text-to-be
       <text
```

where "text-to-evaluat
command (or expressi
(or evaluated) when
procedure's stateme
"success-flag" is a va
used to control recove
EVALed TEXT. EVAL
expression if "text-to
If it is a PROPHET
returned.

The main thing to
the "text-to-be-evalu

the name of a procedure within
t preceded by the word CALL,
voke the procedure named.
g whether a procedure has been
see Section V. above), you
e procedure. You can turn off
of a procedure by prefacing

ROC) THEN . . .

erful function that you can use
ions within a procedure not
system procedures or commands.

f the EVAL function is

evaluated [,success-flag])
<bool>

" is the TEXT of a PROPHET
n) that can be legally executed
ne EVAL is encountered in a
ts. The optional argument
iable of type BOOL that can be
y from a possible failure of the
ill return the value of the
be-evaluated" is an expression.
command, no useful value will be

emember when using EVAL is that
ated" is the TEXT of a command or

expression that can be legally executed or evaluated at top-level PROPHET. That is, any variables referred to by the EVAL TEXT must be global, top-level PROPHET variables. You cannot reference a variable by the name under which it is known within the procedure.

If there is an operation you want to perform for which there are currently no system procedures or commands, EVAL can be used. An example might be the renaming of a PROPHET table. The following sequence of lines from a procedure show how this could be done with an EVAL.

```
    OLDNAME = %GETNAME(DATATAB);

    NEWNAME = %GETTEXT('
       WHAT NEW NAME DO YOU WANT FOR '.OLDNAME,FALSE);

    EVAL('RENAME '.OLDNAME.' TO '.NEWNAME);
```

Note how you can use TEXT variables within the procedure and concatenate them with TEXT constants to produce the EVAL TEXT. Also observe that blank spaces must be provided very carefully so that the EVAL TEXT forms a legal top-level command. Remember that the EVAL TEXT must be a valid command at top-level; thus, the table to be renamed must be referred to by its top-level name, not by its name within the procedure.

The text to be evaluated is not examined when the procedure is compiled; indeed, the entire command represented by the text is compiled at runtime. So you may get either type of error (compilation or execution) when the EVAL is executed. One debugging technique is to assemble "text-to-be-evaluated" into a TEXT variable and to type it out before EVALing it, e.g.

```
    RNMTXT = 'RENAME '.OLDNAME.'TO '.NEWNAME;
    TYPE RNMTXT;
    EVAL(RNMTXT);
```

If, for any reason, the new name you have provided cannot be given to the table, the EVAL will fail and you will be aborted from the procedure. Here is where you can use the optional "success-flag" argument with EVAL to recover from an error and try a new name.

```
    OLDNAME = %GETNAME(DATATAB);

    DO FOREVER;
        NEWNAME = %GETTEXT('
        WHAT NEW NAME DO YOU WANT FOR '.OLDNAME,FALSE);
        EVAL('RENAME '.OLDNAME.' TO '.NEWNAME,OKFLAG);
        IF OKFLAG THEN DOEXIT;

    END;
```

If the EVAL TEXT has been successfuly executed, "success-flag" will be set to TRUE and the procedure exits from the DO . . . FOREVER block; if the EVAL has failed, it will have been set to FALSE and the procedure will ask again for a new name.

C. The BEGIN . . . END Block

The BEGIN . . . END Block is another block structure tool similar to the DO . . . END Block. It differs from the DO . . . END Block in that declarations can be made within a BEGIN . . . END, but not with a DO . . . END. The main use of the BEGIN . . . END Block is in conjunction with the %ERRORFLAG statement described next.

D. The %ERRORFLAG Statement

The %ERRORFLAG statement enables you to control the operation of your program if an error should occur, rather than having the procedure abort. %ERRORFLAG operates in the context of a BEGIN . . . END block. At the beginning of your procedure, you should declare a variable as a BOOL; this will act as the %ERRORFLAG later. Then, you should enclose the possibly troublesome statement in a BEGIN . . . END block along with a statement of the form

```
%ERRORFLAG problem-flag
```

If an error should occur within the BEGIN . . . END Block, the procedure will immediately jump to the END of the BEGIN . . . END block, and the %ERRORFLAG variable will be set to TRUE. Otherwise, the %ERRORFLAG variable will be set to FALSE and execution will continue as usual.

For example, here is part of the definition of a procedure which gets TEXT input and converts it to FLOAT output:

```
DECLARE ERRFLG BOOL;
DECLARE TXTVAL TEXT;
DECLARE X FLOAT;
        .
        .
        .

DO FOREVER;
    TXTVAL = %GETTEXT('
    PLEASE TYPE A NUMBER',FALSE);
    BEGIN;
        %ERRORFLAG ERRFLG;
        X = CFLOAT(TXTVAL);
    END;
    IF NOT ERRFLG THEN RETURN X;
    ELSE TYPE 'PLEASE SUPPLY ANOTHER NUMBER';
END;
```

In this case, if the CFLOAT causes an error (for instance if you typed "Q.2" instead of "3.2"), ERRFLG will be set to TRUE and execution will continue at the IF statement following the BEGIN . . . END block. An error message will be typed and you will have another chance to type a number, since these statements are all enclosed in a DO FOREVER loop. If CFLOAT did not cause an error, ERRFLG will be set to FALSE and the number you typed will be returned.

%ERRORFLAG is particularly useful in conjunction with statements which can cause an error, but which cannot easily be checked out in advance. A good example of this kind of statement is a %MAKETABLE. There is no easy way to check out in advance if the name you have selected for the new table conflicts with an existing table name. Enclosing the %MAKETABLE with a BEGIN . . . END Block and prefacing it with an %ERRORFLAG statement can allow you to catch such errors and take corrective action.

```
DO FOREVER;
    BEGIN;
        %ERRORFLAG MESSEDUPFLAG;
        %MAKETABLE(TAB,TABNAME,'NAMED','PRIVATE');
    END;
    IF NOT MESSEDUPFLAG THEN DOEXIT;
    ELSE DO;
        TYPE NOCR '
THERE IS A PROBLEM NAMING THE TABLE '.TABNAME.'
PLEASE SUPPLY A NEW NAME';
        INPUT TEXT TABNAME;
    END;
END;
```

E. The ERROR START Statement

The ERROR START statement can be used in conjunction with error checking within a procedure to cause an error message to be typed out and to abort the procedure. Often such situations would cause a PROPHET error message to be typed out anyway, but ERROR START allows you to create your own error message. The formal specifications of ERROR START are:

```
ERROR START,error-message;
                <text>
```

where "error-message" is a TEXT constant or a TEXT expression you want typed out. The following example shows a use of ERROR START.

```
DUMMY: PROCEDURE(TAB);
DECLARE TAB TABLE;
          .
          .
          .
IF EMPTY(TAB) THEN ERROR START,'
  YOU MUST CALL DUMMY WITH A NON-EMPTY TABLE!';
```

In the example above, if the table variable TAB was found to be EMPTY, the error message would be typed out and the execution of the procedure aborted.

VII. Table Functions and Procedures

There are a number of PROPHET functions and procedures which allow you to access and to change information stored in tables. These are useful in operating on tables from within procedures since they do not require you to type in any responses from your terminal.

The functions and procedures are summarized in Table 7.1, along with the equivalent commands, where applicable.

A PROPHET table consists of a structure (i.e., the number of columns and rows, title, notes, column headings and types, row names, etc.), and cell data (the values which are contained in the cells of the table). In this section, operations which examine or set cell data will be described first, followed by operations which relate to the structure of an entire table, and then operations which relate to the structure of an individual column.

In all of these functions and procedures, the argument "table" is the name of a variable of type TABLE -- _not_ the TEXT of the table's name. The argument "row-id" can be _either_ the FIXED row number _or_ (if the rows are named) the TEXT of the row name. "Column-id" can be _either_ the FIXED column number, _or_ the TEXT of the column heading.

A. Moving Data To or From a Table Cell

In addition to using the SET command (see Section 4 of the Users Manual) to retrieve or change the data in an individual cell, you can use the %GETCELL function and the %SETCELL procedure. The function

```
value = %GETCELL(table,row-id,column-id)
```

returns the value of the contents of the specified cell.

The procedure

```
%SETCELL(table,row-id,column-id,new-value)
```

will set the contents of the specified cell to "new-value". The data type of "new-value" <u>must</u> match the data type of the column. As with the SET command, you can clear the cell by specifying "new-value" as empty (i.e., by omitting "new-value", or by specifying "new-value" as the name of a variable whose value is empty).

For example, the command

```
CALL %SETCELL(TAB2,K,'RECAP',%GETCELL(TAB1,K,2))
```

would fetch the value from row K, column 2 of table TAB1 and place it in row K of the column in TAB2 whose heading is RECAP.

B. Operations on the Structure of a Table

Table Name

The function

```
directory-name = %GETNAME(table)
```

returns the TEXT of the name under which the table is known in its directory. The argument is the variable name under which the table is available to you. This function is useful, for example, if you have used the AS option in an ACCESS command or if you want to use the table's name to perform some textual manipulation.

Table Title

The function

```
title = %GETTITLE(table)
```

returns the TEXT of the table's title. You can change the title by using the procedure

```
%SETTITLE(table,new-title)
```

which will change the title of the table to the TEXT specified in "new-title". You can clear the title if "new-title" is empty or the null text('').

Table Type

The function

```
table-type = %GETTYPE(table)
```

returns a TEXT value which is 'PRIVATE', 'SHARABLE', 'PUBLIC', 'TEMPORARY' or 'SHADOW', depending on the table type. (See Chapter IV of the Users Manual for a description of these table types.)

Number of Rows and Columns

Two functions are useful for determining the number of rows or columns in a table. The function

```
last-col-no = %LASTCOL(table)
```

returns the FIXED value of the number of the last column in the table. Note that this number is the same, regardless of whether or not the table has data in column 0, i.e. has row names. The function

```
row-size = %LASTROW(table)
```

returns a FIXED value which is the number of rows in the table. These two functions can be used in a procedure which operates on a table, for example, when you want to determine an upper limit for a DO Loop.

Row Names

There are two functions for determining whether or not a table has row names. The function

```
row-name-text = %GETRNAME(table)
```

returns either the TEXT 'NAMED' or the TEXT 'UNNAMED' depending on whether or not the table has row names. The function

```
begin-col = %FIRSTCOL(table)
```

returns a FIXED value which is 0 if the rows are named, and 1 if the rows are not named (if a table has row names, they are stored in column 0). %FIRSTCOL is useful when you want to determine the lower limit for a DO Loop which operates on the contents of each cell of a table row. For example,

```
DO I = %FIRSTCOL(TAB2) TO %LASTCOL(TAB2);
```

will cause the DO Loop to be executed beginning with column 0 (which contains the row names) if the rows are named, and with column 1 if the rows are not named.

You can alter a table's capability of accommodating row names by using the procedure

```
%SETRCOL(table,column-number)
```

where "column-number" is set to 0 to add the row name capability, and is EMPTY to delete it. For example, the procedure call

```
CALL %SETRCOL(TAB2,0)
```

will give the table TAB2 the capability of having row names. The procedure calls

```
CALL %SETRCOL(TAB2,)      or
CALL %SETRCOL(TAB2,VEM)
```

where VEM is an empty FIXED variable, removes TAB2's capability of having rows referred to by name. However, if there are entries in column 0, they are not deleted. The row name capability can be restored later and the existing row names will again be available for use.

Table Notes and Footnotes

The function

```
table-notes = %GETNOTES(table)
```

returns the TEXT of the notes field of the table. You can change the contents of the notes field by using the procedure

```
%SETNOTES(table,new-notes)
```

which will change the notes to the TEXT specified in "new-notes".

There are analogous functions and procedures to handle the footnotes of a table. The function

```
footnote = %GETFOOTNOTE(table,footnote-index)
                                <fixed>
```

returns the TEXT of the footnote specified by the FIXED argument "footnote-index". You can change a specified footnote by using the procedure

```
%SETFOOTNOTE(table,footnote-index,new-footnote)
```

where "footnote-index" is the FIXED number of the footnote you want to set, and "new-footnote" is the TEXT you want the specified footnote to have.

It is also possible to access or change all of a table's footnotes at one time. The function

```
notes-array = %GETALLFOOTNOTES(table)
```

returns a TEXT array containing all footnotes of the table. The procedure

```
%SETALLFOOTNOTES(table,notes-array)
```

gives the table the footnotes specified in the TEXT array "notes-array".

C. Operations on the Structure of a Column

There are several procedures which may be used to access or change the structure of a particular column of a table.

Column Headings

The function

```
col-head = %GETCHEAD(table,column-id)
```

returns the TEXT of the specified column's heading. The procedure

```
%SETCHEAD(table,column-id,new-heading)
```

sets the heading of the specified column to the TEXT given in "new-heading". If the "new-heading" would not be unique for that table, the procedure will give an error.

A heading can be "tried out" first in order to assure that it will be a unique column heading by using the function

```
col-head = %UNIQUECHEAD(table,new-heading)
```

which returns a TEXT guaranteed to be a unique column heading for the table. If the heading specified in "new-heading" is unique to start with, it will be returned intact; but if there is already a heading in the table that is the same as "new-heading", period (".") will be appended to "new-heading" to make it unique.

Column Type

The function

```
col-type = %GETCTYPE(table,column-id)
```

returns the TEXT of the specified column's data type. The procedure

```
%SETCTYPE(table,column-id,new-type)
```

sets the data type of the specified column to the data type specified by the TEXT argument "new-type". This procedure may be used only with columns which have no data in them or where the data are compatible with the new type.

Column Derivations

The function

```
col-deriv = %GETCDERIV(table,column-id)
```

returns the TEXT of the specified column's derivation. The procedure

```
%SETCDERIV(table,column-id,new-derivation)
```

sets the derivation of the specified column to the TEXT given in "new-derivation". (See Chapter IV of the Users Manual for a discussion of the format and restrictions that apply to derivations.) If you put the actual text of the derivation in the procedure call, it must be enclosed in single quotation marks. For example,

```
CALL %SETCDERIV(ABC,4,'3.*CELL(2)')
```

will set the value in each cell of column 4 of table ABC to three times the value in the corresponding cell in column 2. Specifying a null-text ('') will clear the derivation, but will not change the actual data in the column (which may have been generated by a previous derivation):

```
CALL %SETCDERIV(ABC,4,'')
```

Used in this way, %SETCDERIV causes the column to no longer be derived, and values may be set into its cells.

If you use a derivation with the column *heading* (instead of the column *number*) in the cell reference, then that column heading must be enclosed in two pairs of single quotation marks (i.e., in order to enclose single quotation marks within single quotation marks, the inner quotation marks must be doubled). For example, if column 2 has the heading RPM/CC, then the above command could take the form

```
CALL %SETCDERIV(ABC,4,'3.*CELL(''RPM/CC'')')
```

Note that you use a pair of single quotation marks, '', *not* a double quotation mark, ". The same derivation entered within the MAKE TABLE dialogue would look like

```
ENTER DERIVATION: 3.*CELL('RPM/CC')
```

This procedure then performs the operation described by the new derivation so that the column contains the correct new data.

Using Descriptors

The function

```
desc-list = %GETCDESC(table,column-id)
```

returns a TEXT which is a list of the descriptor values (separated by spaces) associated with the specified column. You can also access the descriptor values as a TEXT ARRAY, rather than as a list, by the function

```
desc-array = %GETCDESCARRAY(table,column-id)
```

which returns a TEXT array containing the descriptors of the specified column. For example, if column 3 of a table named DATA was of type DESCRIPTOR, and you wanted to put the descriptor list in a TEXT array named DESCLIST, you could use these commands:

```
DECLARE DESCLIST TEXT ARRAY
DESCLIST = %GETCDESCARRAY(DATA,3)
```

The procedure

```
%SETCDESC(table,column-id,new-desc-list)
```

sets the descriptor list of the specified column to the TEXT given as "new-desc-list". Each descriptor in the list must be separated by one or more spaces. The procedure

```
%SETCDESCARRAY(table,column-id,new-desc-array)
```

sets the descriptor list of the specified column to the TEXT array given as "new-desc-array". With both of these procedures, if there are any data already in the column, they must be consistent with the new descriptor list.

Column Width and Format

The width of a column can be obtained by the function

```
col-width = %GETCWIDTH(table,col-id)
```

which returns the FIXED width (measured as the number of characters that will be shown in each column when the table is displayed) of the specified column.

You can change the width of a column by using the procedure

```
%SETCWIDTH(table,col-id,new-width)
```

where "new-width" is the FIXED number of characters you want the column to accommodate in a display.

A column can also have a format associated with it which causes the data in the column to be displayed in a specified manner. (See Section 4 of the Users Manual for a detailed description of the various formats which can be used.) The function col-format = %GETCFORMAT(table,col-id)

returns the TEXT of the specified column's format.

A column can be given a new format by using the procedure

%SETCFORMAT(table,col-id,new-format)

sets the format of the data in the specified column to the format designated by the TEXT in "new-format".

D. Making Tables Within a Procedure

There are two basic ways to create a new table within a procedure: making a completely new table or making a table from an existing table.

1. Making a Table from Scratch

If you want to create a completely new table, you should use the procedure

```
%MAKETABLE(local-name,global-name,rowname-spec.table-type)
           <table>     <text>      <text>       <text>
```

In the %MAKETABLE procedure, "local-name" is a local variable of type table; "global-name" is the TEXT of the name you want the table to have at top-level PROPHET (it is the equivalent of the name you would use when you use the "MAKE TABLE" command); "rowname-spec" is a TEXT which can have the values 'NAMED' or 'UNNAMED', depending on whether or not you want the table to have row names; "table-type" is a TEXT which determines what kind of table is being created, it can have the values: 'PRIVATE', 'SHARABLE', 'PUBLIC', 'TEMPORARY' or 'SHADOW'. A TEMPORARY table exists only within your workarea; it is not automatically saved for you from session to session as are all the other types of tables. Remember that only tables small enough to fit in your workarea may be TEMPORARY tables. A SHADOW table is a special kind of TEMPORARY table; if the TEMPORARY table you want to make is too large to fit in your workarea, you should make a SHADOW table. Like the TEMPORARY table, the SHADOW table will not be saved from session to session.

A table created with %MAKETABLE will be completely empty of data, and will have zero rows and zero columns. To establish columns, the %ADDCOLUMN procedure described below may be used.

2. Making a Table from another Table

If you want to create a new table which contains all or part of the data from an existing table, you can use a non-interactive form of the MAKE TABLE command

```
MAKE [type] TABLE local-name NAMED global-name [options]
```

where the optional "type" can be PRIVATE, SHARABLE, or so forth; where "local-name" is the local variable name by which you will be referring to the table within the procedure (it must previously have been declared within the procedure to be a variable of type TABLE); and where "global-name" is the name of the table's directory entry, that is, the name by which the table will be known outside the procedure. (In the case of a TEMPORARY table, there will be no directory entry, but "global-name" will still be the name by which the table will be known outside the procedure.) "Global-name" must be either a constant TEXT (i.e., enclosed in single quotes), or a TEXT variable. Using a TEXT variable as "global-name" is useful, for example, if you have computed a TEXT within a procedure and want to create a global table with that name. "Options" may be any of the options available with the MAKE TABLE command; e.g., USING, FROM, AS DISTRIBUTION OF, etc.

For example, the command

```
MAKE TABLE LOCALTAB NAMED 'GLOBALTAB'
    FROM NOTHERGLOBALTAB FOR COL 1 > 10.
```

within a procedure will create the same table which would have been produced by the top level command

```
MAKE TABLE GLOBALTAB FROM NOTHERGLOBALTAB
    FOR COL 1 > 10.
```

However, the (local) variable LOCALTAB should be used to reference the new table from within the procedure.

E. Adding Columns to Tables

To add a column structure to a table, the procedure

```
%ADDCOLUMN(table,heading,type)
                 <text> <text>
```

may be used. "Heading" is the TEXT of the column heading. Remember that each column within a table _must_ be unique; it is a good idea when using %ADDCOLUMN in a procedure to use the %UNIQUECHEAD function described above to assure that the heading to be added is truly unique.

The third argument, "type", is a TEXT naming the type of data the column will contain. It can be the name of any legal data type, e.g., 'NUMBER', 'TEXT', 'CHEMTEXT', 'GRAPH', 'MOLECULE', 'DESCRIPTOR', etc. If the column is of type 'DESCRIPTOR', the procedure %SETCDESC (or %SETCDESCARRAY), described above, must be used to specify the descriptor list. If the column is to have derived data, then the procedure %SETCDERIV, described above, can be used to specify the derivation.

The %ADDCOLUMN procedure specifies only the structure of the new column; to place data into the cells of the column, you must use the array-to-table transfer procedures described below, or the ENTER COLUMN[S] or FILLIN commands described in Chapter IV of the Users Manual.

Inserting
Columns
or Rows

%ADDCOLUMN always adds a column _after_ the last existing column in the table. To insert a new column _between_ two existing columns of a table, use the procedure

```
%INSERTCOLUMN(table,column-id,heading,type)
```

The new column will be inserted at the column specified in "column-id". For example, if you wanted to insert a new column in table DATATAB between the existing columns 2 and 3, and you want the new column to be of type NUMBER with a heading of 'NEWDATA', the command

```
CALL %INSERTCOLUMN(DATATAB,3,'NEWDATA','NUMBER')
```

would create a new column 3 (empty of data) with the heading NEWDATA. The old column 3 would now be column 4, and so forth.

Analogously, the procedure %INSERTROW can be used to insert a new row between two existing rows of a table:

```
%INSERTROW(table,row-id)
```

where "row-id" is either the row number or the row name of the row at which you wish to insert the new row.

F. Allocating Space for Tables

Whenever the number of rows and/or columns of a table grows by more than a certain amount, the computer must re-allocate space for the table. Ordinarily, PROPHET allocates additional space in reasonably-sized increments; however, if you are adding more than about 20 rows or columns to a table, and if you know in advance how large the table will be, you can increase PROPHET's efficiency by using the statement

```
ALLOCATE table(total-rows,total-columns)
                <fixed>      <fixed>
```

to pre-allocate the required space. This only *allocates* the additional space; no data or structure is added to the table.

If the ultimate dimensions of a table are known when you are using the %MAKETABLE procedure, you can combine the table-making with an allocation by using the procedure

```
%MAKETABLEALLOC(local-name,global-name,rowname-spec,
                 <table>     <text>     <text>

             table-type,total-rows,total-cols)
               <text>    <fixed>     <fixed>
```

which is identical to %MAKETABLE except for the two additional FIXED arguments "total-rows" and "total-cols" which are used to pre-allocate the table being made to the appropriate size.

G. Deleting Columns, Rows, and Tables

To delete a column from a table, you can use the procedure

```
%DELETECOLUMN(table,column-id)
```

If there are any derived columns which use the column being deleted, the system will type the message

```
COLUMNS DERIVED FROM COLUMN column-id ARE:

first-derived-column, . . . , last-derived-column

DO YOU STILL WANT TO DELETE IT?
```

If you answer NO, then the column will not be deleted. If you answer YES, then the column is removed, and all subsequent columns are renumbered; in addition, the derivations of the dependent columns (but not the dependent columns themselves or the data they contain) are deleted. This has the same effect on the derived columns as if you had used the procedure %SETCDERIV (described above) with a null TEXT, '', as the third argument on each of them.

To delete a row from a table, you can use the procedure

```
%DELETEROW(table,row-id)
```

This removes the row from the table and renumbers all subsequent rows.

An entire table can be deleted from within a procedure by using the procedure

```
%DELETETABLE(table)
```

In this procedure <u>only</u>, the argument can be <u>either</u> a variable of type TABLE, <u>or</u> a variable of type TEXT which contains the name of the table to be deleted.

H. Table/Array Data Transfer

There are many times when you will want to perform operations on data in a table. When the number of data elements is large, or you want to perform many operations on them, it is often more efficient to perform the operations on arrays of data rather than on the cells of the table. The procedures described below allow you to tranfer table data to arrays and vice versa. (Declaring and using arrays is discussed above in Section I.E)

These procedures can either transfer a portion of the data from a table into a two-dimensional array (and the reverse operation) or transfer all or part of the data from a single column or row of a table into a one-dimensional array (and the reverse). In performing some operations on cell data, it may be faster to transfer the data from the table into an array, perform the operation on the array and transfer the data back into the table, than it is to operate directly on the data in the table.

It is important to remember that those procedures which transfer data into an array from a table, namely, %T2A, %C2A, and %R2A, clear the argument array before reading from the table. Therefore, any data you put into the array before calling one of these procedures will be lost, even when it is in a part of the array that would not be utilized for the table data.

Two-dimensional Arrays

The procedure to transfer data from some portion of a table into a two-dimensional array is

```
%T2A(table,column-specs,first-row-id,last-row-id,
      array-name,transpose-option)
```

"Column-specs" is either a FIXED or TEXT array or array constant which contains the column-ids to be used; they need not be contiguous nor in order (and a column may be mentioned more than once), but the data types of the columns must match the data type of the array. If you want all the columns of the table, each one must be identified in the array. There is no shorthand for referring to the complete set in columns of a table. The next two arguments, "first-row-id" and "last-row-id", are either FIXED or TEXT and are the first and last row-ids to be included in the transfer; the rows, therefore, must be contiguous. (If "first-row-id" is left empty, 1 is used; if "last-row-id" is empty, the last row of the table is used.) The argument "array-name" is the name of the two-dimensional array into which the table data are to be transferred. It must be of the proper data type to accept the data in the table. (That is, if all of the columns are of type NUMBER, the array can be a two-dimensional FLOAT array, but if there are mixed data types, the array must be declared to be a two-dimensional ANY array.) Finally, "transpose-option" is a TEXT variable or constant; if it has the value 'TRANSPOSE', the rows of the table will become columns of the array, and the columns, rows (for example, without the 'TRANSPOSE' option, the array notation A(I,J) references the array datum which was transferred from row I, column J of the table; with the 'TRANSPOSE' option, A(I,J) references the datum in row J, column I of the table). If "option-text" has any other value at all, e.g. '', then A(I,J) is set to row I column J of the table.

Thus, if you had a table named DATATABLE, composed of 5 columns (all of type NUMBER) and 10 rows, and you wanted to transfer the data into a two-dimensional FLOAT array, you could use the following commands:

```
DECLARE DATARA(*,*) FLOAT
CALL %T2A(DATATABLE,[1,2,3,4,5],1,10,DATARA,'')
```

In order to transfer data from a two-dimensional array into a portion of a table, the corresponding procedure

```
%A2T(table,column-specs,first-row-id,last-row-id,
      array-name,transpose-option)
```

is used, where the arguments have exactly the same meaning as before.

In the argument lists to the procedures %T2A and %A2T, only "table" and "array-name" need actually be variable names. All the other arguments may be constants of the proper type. In using %A2T, it is necessary to have an already defined table structure available (with appropriate columns) into which the array data is to be transferred (i.e., the columns must have already been created with headings and types; the rows need not have already been created).

One-dimensional Arrays

The procedure to transfer data from a single column of a table into a one-dimensional array is

```
%C2A(table,column-id,first-row-id,last-row-id,array-name)
```

The procedure to transfer data from a one-dimensional array into a column of a table is

```
%A2C(table,column-id,first-row-id,last-row-id,array-name)
```

Here the arguments have exactly the same meanings as for the two-dimensional cases except that "column-id" is now a single FIXED or TEXT column-id. In both %A2C and %C2A, the rows in the transfer must be contiguous. In %A2C the data in the array must match the data type of the column; in %C2A, the array must be of the proper type to accept data from the column of the table.

The procedure to transfer data from a single row of a table into a one-dimensional array is

```
%R2A(table,row-id,first-col-id,last-col-id,array-name)
```

The procedure to transfer data from a one-dimensional array into a row of a table is

```
%A2R(table,row-id,first-col-id,last-col-id,array-name)
```

The columns used must be contiguous in the table. In addition, all the columns must contain compatible data types.

I. Miscellaneous Table Functions and Procedures

Other table functions and procedures include:

1. %TESTPROTECT

As described in Section 4 of the Users Manual, tables can be protected against SYSTEM, PROJECT, and SELF. If you want to determine if a table has been protected, you can use the function

```
prot-flag = %TESTPROTECT(table,prot-type)
  <bool>                          <text>
```

which returns TRUE if the table is protected against the TEXT argument "prot-type" and FALSE if the table is not so protected. "Prot-type" should be either 'SYSTEM', 'PROJECT', or 'SELF'.

2. %DIRECTORYCHECK and %ARCHIVECHECK

You may sometimes want to check and see if a table already exists which has the same name as one you are about to make. The function

```
find-flag = %DIRECTORYCHECK(dir-type,table-name)
  <bool>                     <text>    <text>
```

returns TRUE if the TEXT "table-name" is found in the specified directory -- "dir-type" must be 'PRIVATE', 'SHARABLE', or 'PUBLIC' -- and FALSE if the name is not found.

Similarily, the function

```
find-flag =  %ARCHIVECHECK(archive-type,table-name)
  <bool>                      <text>       <text>
```

returns TRUE if "table-name" is found in the archive directory specified by the TEXT "archive-type", and FALSE if it is not found.

TABLE 7-1: TABLE PROCEDURES AND FUNCTIONS

	Procedure or Function	Procedure or Function Name and Arguments
1	Function (RETURNS ANY)	%GETCELL(table,row-id,col-id)
2	Procedure	%SETCELL(table,row-id,col-id,new-value)
3	Function (RETURNS TEXT)	%GETNAME(table)
4	Function (RETURNS TEXT)	%GETTITLE(table)
5	Procedure	%SETTITLE(table,new-title)
6	Function (RETURNS TEXT)	%GETTYPE(table)
7	Function (RETURNS FIXED)	%LASTCOL(table)
8	Function (RETURNS FIXED)	%LASTROW(table)
9	Function (RETURNS TEXT)	%GETRNAME(table)
10	Procedure	%SETRCOL(table,col-number)
11	Function (RETURNS FIXED)	%FIRSTCOL(table)
12	Function (RETURNS TEXT)	%GETNOTES(table)
13	Procedure	%SETNOTES(table,new-notes)
14	Function (RETURNS TEXT)	%GETFOOTNOTE(table,footnote-index)
15	Procedure	%SETFOOTNOTE(table,footnote-index,new-footnote)
16	Function (RETURNS TEXT ARRAY)	%GETALLFOOTNOTES(table)

TABLE 7-1: TABLE PROCEDURES AND FUNCTIONS

	Command Syntax
1	COLUMN col-id ROW row-id OF table
2	SET COLUMN col-id ROW row-id OF table TO new-value
3	
4	TITLE of table
5	SET TITLE OF table TO new-title
6	
7	NUMBER OF COLUMNS OF table
8	NUMBER OF ROWS OF table
9	
10	
11	
12	NOTES OF table
13	SET NOTES OF table TO new-notes
14	
15	
16	

TABLE 7-1: TABLE PROCEDURES AND FUNCTIONS

	Procedure or Function	Procedure or Function Name and Arguments
17	Procedure	%SETALLFOOTNOTES(table,notes-array)
18	Function (RETURNS TEXT)	%GETCHEAD(table,col-id)
19	Procedure	%SETCHEAD(table,col-id,new-heading)
20	Function (RETURNS TEXT)	%GETCTYPE(table,col-id)
21	Procedure	%SETCTYPE(table,col-id,new-type)
22	Function (RETURNS TEXT)	%GETCDERIV(table,col-id)
23	Procedure	%SETCDERIV(table,col-id,new-derivation)
24	Function (RETURNS TEXT)	%GETCDESC(table,col-id)
25	Procedure	%SETCDESC(table,col-id,new-desc-list)
26	Function (RETURNS TEXT ARRAY)	%GETCDESCARRAY(table,col-id)
27	Procedure	%SETCDESCARRAY(table,col-id,new-desc-array)
28	Function (RETURNS TEXT)	%GETCFORMAT(table,col-id)
29	Procedure	%SETCFORMAT(table,col-id,new-format)
30	Function (RETURNS FIXED)	%GETCWIDTH(table,col-id)
31	Procedure	%SETCWIDTH(table,col-id,new-width)
32	Procedure	%MAKETABLE(local-name,global-name, rowname-spec,table-type)

TABLE 7-1: TABLE PROCEDURES AND FUNCTIONS

	Command Syntax
17	
18	HEADING OF COLUMN col-id OF table
19	SET HEADING OF COLUMN col-id OF table TO new-heading
20	TYPE OF COLUMN col-id OF table
21	SET TYPE OF COLUMN col-id OF table TO new-type
22	DERIVATION OF COLUMN col-id OF table
23	SET DERIVATION OF COLUMN col-id OF table TO new-derivation
24	DESCRIPTOR OF COLUMN col-id OF table
25	SET DESCRIPTOR OF COLUMN col-id OF table TO new-desc-list
26	
27	
28	FORMAT OF COLUMN col-id OF table
29	SET FORMAT OF COLUMN col-id OF table TO format-spec
30	WIDTH OF COLUMN col-id OF table
31	SET WIDTH OF COLUMN col-id OF table TO new-width
32	MAKE TABLE local-name NAMED global-name [options]

TABLE 7-1: TABLE PROCEDURES AND FUNCTIONS

	Procedure or Function	Procedure or Function Name and Arguments
33	Procedure	%MAKETABLEALLOC(local-name,global-name, rowname-spec,table-type,total-rows,total-cols)
34	Procedure	%ADDCOLUMN(table,heading,type)
35	Procedure	%INSERTCOLUMN(table,col-id,heading,type)
36	Procedure	%INSERTROW(table,row-id)
37	Procedure	%DELETECOLUMN(table,col-id)
38	Procedure	%DELETEROW(table,row-id)
39	Procedure	%DELETETABLE(table)
40	Procedure	%T2A(table,col-specs,first-row, last-row,array-name,transpose-option)
41	Procedure	%A2T(table,col-specs,first-row, last-row,array-name,transpose-option)
42	Procedure	%C2A(table,col-id,first-row, last-row,array-name)
43	Procedure	%A2C(table,col-id,first-row, last-row,array-name)
44	Procedure	%R2A(table,row-id,first-col, last-col,array-name)
45	Procedure	%A2R(table,row-id,first-col, last-col,array-name)

TABLE 7-1: TABLE PROCEDURES AND FUNCTIONS

	Command Syntax
33	
34	ADD COLUMNS TO table
35	INSERT COLUMN col-id OF table
36	INSERT ROW row-id OF table
37	DELETE COLUMN col-id OF table
38	DELETE ROW row-id OF table
39	DELETE table
40	
41	
42	
43	
44	
45	

VIII. Displ

Although the dis
adequate for most
want to generate a s
The procedures des
do a certain amount
fairly primitive,
to specify location
described as argu
coordinates. Any d
display routines
will erase the disp
noise occurs while
will not regenerate
entire display.

Screen Geometry

The display scr
rectangle. The or
the center of the s
-512 (at the le
coordinates run fr
the top). Screen
not cause an er
specified display
A single character
box 14 X units wid
portion of the sc
at the top of the
for displaying
command portion is
22 Y units for eac
is set to 2, th
114 Y units, and
units.

y Procedures and Functions

lay part of PROPHET is generally
users' needs, you may occasionally
pecial display within a procedure.
cribed in this chapter allow you to
of special formatting. They are
nd, among other things, require you
on the screen. All coordinates
ents to these procedures are screen
splay which you create using these
s very ephemeral; a CLEAN command
ay so created. If, therefore, line
our display is being drawn, a CLEAN
it. You must instead redraw the

een of the Tektronix terminal is a
igin of the screen coordinates is at
creen. The X coordinates run from
t) to +511 (at the right). The Y
m -390 (at the bottom) to +389 (at
oordinates outside these ranges will
or in the display procedures; the
imply will not appear on the screen.
on the screen will be displayed in a
and 22 Y units high. The command
een occupies a minimum of 70 Y units
creen. Thus, the available Y range
s not more than 710 Y units. If the
set to other than zero, you must add
n line, i.e. if the command portion
command portion will occupy at least
ne available display portion is 666 Y

Note in all the procedures described below that the arguments designating screen coordinates may be either FIXED or FLOAT.

Simple Drawing

To draw a single point on the screen, use the procedure

    %DPOINT(x-coord,y-coord)

This will draw a bright point on the screen at the coordinates given. If the coordinates are not within the range of the screen, no visible result will occur.

To draw a single straight line on the screen, use the procedure

    %DLINE(point1-x,point1-y,point2-x,point2-y)

This will draw a line from the X,Y coordinates given as the first two arguments to the X,Y coordinates given as the last two arguments. If either one of the pair of X,Y coordinates is not within the range of the screen, the line will run off one edge. If both pairs of coordinates are not within the range of the screen, you will not see anything.

To draw a rectangle on the screen, use the procedure

    %DBOX(left-x,top-y,right-x,bottom-y)

This will draw a rectangle whose upper left hand corner is located at the X,Y coordinates given by the first two arguments, and whose lower right hand corner is located at the X,Y coordinates given by the last two arguments. If any of the four coordinates is not within the range of the screen, as much of the box as possible will be drawn, but some of the lines defining the box will run off the edge of the screen.

Displaying a Text

In order to display a TEXT or a numeric variable somewhere on the screen, use the procedure

%DTEXT(var-name,way-text,left,top,right,bottom)

This procedure displays the first argument (a numeric or TEXT variable or constant) in the box defined by the X,Y coordinates given as the last four arguments. Note that the box itself is *not* displayed. These four coordinates have the same meaning as the arguments to %DBOX. The text may contain any characters whatever, including carriage returns (produced by the RETURN key on your terminal). A single line of a text is ended by the occurence of a carriage return. "Way-text" is a TEXT which may have one of two legal values. If the value of "way-text" is 'CR', any line of the text which is too long to fit in the box defined, will be continued on the next line, i.e. a free carriage return is inserted at the right boundary of the box. If "way-text" has the value 'NCCR', any line which is too long to fit in the box defined will be truncated at the right boundary of the box. Note that "way-text" only operates for the X direction. If the text has too many lines to fit in the box defined in the Y direction, the extra lines will not be displayed. If you only wish to specify the upper left hand corner of the box in which the text is to be displayed, the last two arguments should be outside the range of the screen, i.e., "right" should be greater than 512 and "bottom" should be less than -390. When the text is displayed within the box, the upper left hand corner of the 14 * 22 cell containing the first character of the text will start 1 Y unit down and 9 X units to the right of the upper left hand corner of the box. One additional Y unit must be added to the height of the box to display the text successfully. If the text to be displayed consists of the characters

THIS IS A TEXT

the box to contain it must be a minimum of 205 (14 chars * 14 units/char + 9 units) X units long and 24 (1 char * 22 units/char + 2 units) Y units high. If the text to be displayed contains a carriage return, each carriage return will require an additional 22 Y units. For example, the text

THIS IS A TEXT
IT IS IN A BOX

will require a box at least 205 X units long and 46 Y units high. The various distances (not to scale) are shown in Figure 8.1. If the text consists of

THIS IS A TEXT IT IS IN A BOX

with no embedded carriage return, it may be displayed in its entirety in two ways. You may give it a box 415 X units (29 chars * 14 units/char + 9 units) and 24 Y units. Or, if "way-text" has the value 'CR', it will also fit in a box 219 X units (15 chars * 14 units/char + 9 units) and 46 Y units.

The procedure %DTEXT can be used to display numbers as well as text. In order to do this, you may call %DTEXT with a FIXED or FLOAT number as the first argument.

Determining the Size of a Large Text

For a text which contains only a few characters, one can simply count characters to find out the size box needed to display it. For a text which is lengthy, it is useful to have a procedure which will do the size computation for you. The procedure

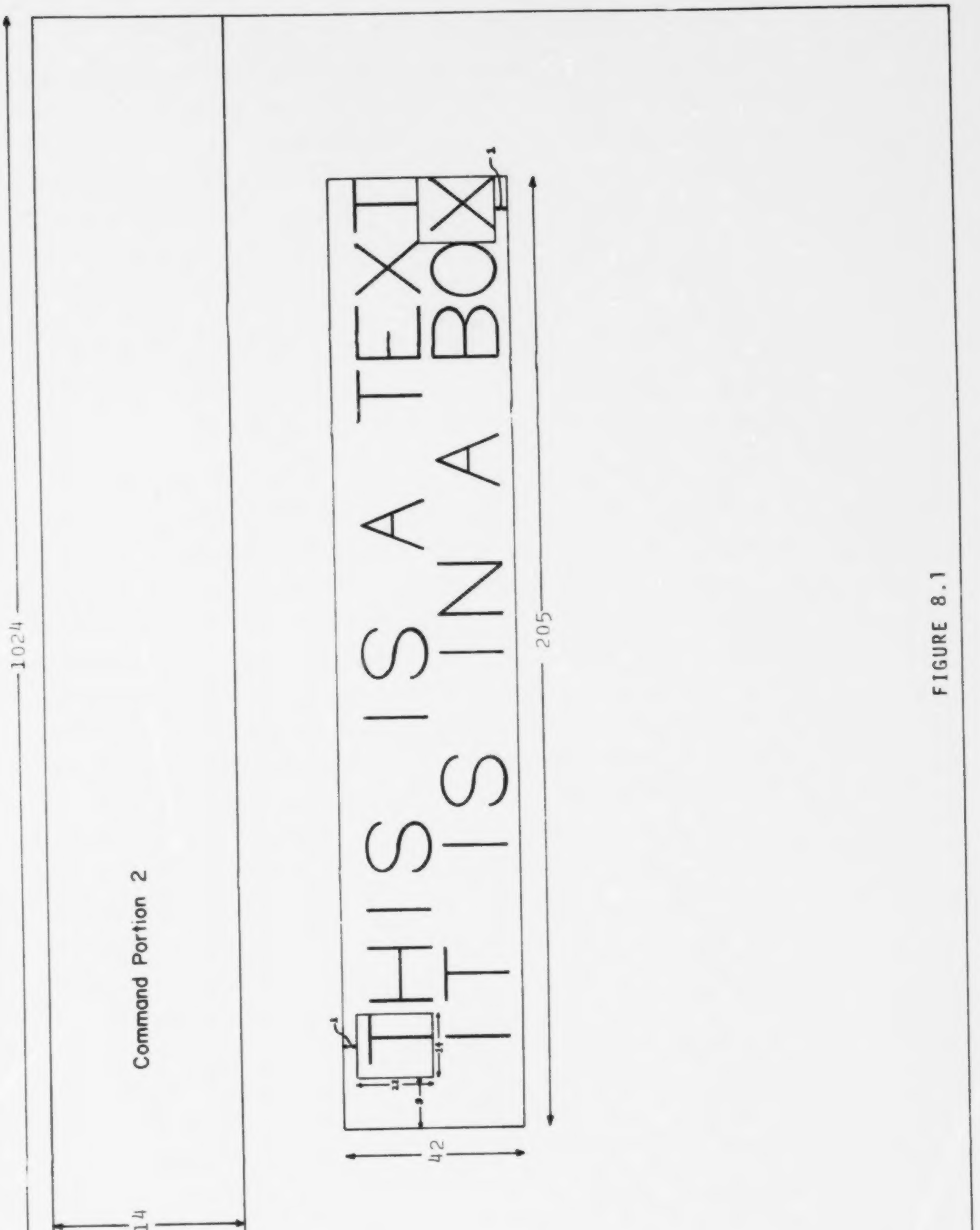

Command Portion 2
THIS IS A TEXT
IT IS IN A BOX

FIGURE 8.1

%DMEASURE(text-name,way-text,x-size,y-size)

takes the TEXT variable or constant to be measured as the first argument. The length of the text, in X units is returned as the third argument, "x-size"; the height of the text, in Y units, is returned as the value of "y-size". The width includes not only the 9 X units required for the left margin, but an additional 9 X units for a right margin. The height includes not only the 1 Y unit required for the top margin, but an additional 6 Y units for a bottom margin. These values, when used in a call to %DTEXT produce a text which looks centered in the box described.

The second argument, "way-text", may have the same two values as that described for it in %DTEXT. If the value of "way-text" is 'NOCR', %DMEASURE will return the actual maximum length of the text. When the text is displayed with %DTEXT, the same value of "way-text" ('NOCR') should be used, or there may be insufficient Y space allotted. If the text is in fact too long for the screen, some of it will not be displayed. If the value of "way-text" is 'CR', the procedure %DMEASURE will insert a free carriage return when the number of characters across (not including margins) would require 854 X units (61 characters). Additional Y units are then included for each extra line thus generated. Again, if the text is measured with "way-text" equal to 'CR', %DTEXT should be called with the same value. These comments clearly only apply to texts which are likely to be longer than the width of the screen. If the text contains embedded carriage returns such that no line is longer than the available X space on the screen, the value of "way-text" in %DMEASURE is irrelevant.

Locating Display Coordinates

When using the display procedures described in this section, it is occasionally useful to obtain the display coordinates of an object displayed on the screen. Procedures exist to do this for displayed graphs and displayed molecules. (For molecules see Section 9.C below).

To determine the display coordinates of a point on a graph, use the function

```
flag = %GRAPHCOORDS(graph,axis,graph-coord,display-coord)
```

The first argument is the graph, which must be displayed on the screen when %GRAPHCOORDS is called. "Axis" is a TEXT designating the axis you wish and must have the value 'X', 'Y', or 'R'. "Graph-coord" is a FLOAT which contains the graph-coordinate whose display-coordinate you wish to obtain, and returned in the last argument is a FLOAT value for the corresponding display-coordinate. The function will return the value TRUE if the graph-coordinate supplied is within the range of the axis specified, and FALSE otherwise. If the supplied coordinate is out of range, the last argument will contain the display coordinate of the end of the designated axis (in the direction of the supplied coordinate). In order to use this procedure to put an additional point on the screen, you must call it twice, once for the X-coordinate and once for the Y-coordinate. If you call the procedure twice for X and twice for Y, you will then have the information needed to set up the transformation function between linear graph-coordinates and display-coordinates.

Miscellaneous Display Procedures

There are two remaining display procedures which allow you to control the terminal components. The first is

```
%DERASE
```

This procedure erases the terminal screen, including the command portion. If an ordinary display was on the screen at the time %DERASE was called, it will be regenerated by a subsequent CLEAN command.

The procedure

```
%DCOPY
```

activates the hard-copier to copy the contents of the screen (this is the same as pressing the hard copy button).

Example

In order to illustrate how these procedures might be used in drawing something on the terminal screen, an illustrative example is given here. Comments are shown in lower case for easier reading. This procedure is currently available as a PUBLIC SAVED variable named POLYGON. If you want to see how the actual display looks, you may RESTORE and try running it.

```
POLYGON:PROCEDURE(N,CTRX,CTRY,DIAM,MESS);

/* Procedure to draw an n-sided regular polygon and
place an arbitrary message in the center of it.

Arguments:
N           number of sides in polygon
CTRX        X coordinate of center of polygon
CTRY        Y coordinate of center of polygon
DIAM        size of polygon (actually diameter
            of superscribed circle)
MESS        label to place at center of polygon */

DECLARE (CTRX,CTRY,DIAM) FLOAT;
DECLARE N FIXED;
DECLARE MESS TEXT;
DECLARE (RAD,ANG,DELANG,XOLD,YOLD,XNEW,
        YNEW,W,H,LL,RR,BB,TT) FLOAT;
DECLARE I FIXED;

/* Check validity of N. */
    IF N<3 THEN DO;
        TYPE 'POLYGON MUST HAVE AT LEAST 3 SIDES';
        ABORT;
    END;

/* Get a clean screen */

    CALL %DERASE;

/* Set up for polygon loop */

    RAD=.5*DIAM;
    DELANG=360./N;

/* Start at top of figure */

    ANG=90.;
    XOLD=CTRX;
    YOLD=CTRY+RAD;

/* Loop to draw polygon */

    DO I=1 TO N;
        ANG=ANG+DELANG;
        XNEW=CTRX+RAD*COSD(ANG);
        YNEW=CTRY+RAD*SIND(ANG);
        CALL %DLINE(XOLD,YOLD,XNEW,YNEW);
        XOLD=XNEW;
        YOLD=YNEW;
    END;
```

```
/* Is there a message to be displayed? */

    IF EMPTY(MESS) THEN RETURN;
       ELSE IF MESS='' THEN RETURN;

/* Measure the message */

    CALL %DMEASURE(MESS,'CR',W,H);

/* Display it in a box centered at CTRX,CTRY */

    LL=CTRX-W/2.;
    RR=CTRX+W/2.;
    BB=CTRY-H/2.;
    TT=CTRY+H/2.;

    CALL %DTEXT(MESS,'CR',LL,TT,RR,BB);

/* Make a hard copy of the screen */

    CALL %DCOPY;

    END;
```

XI. Communicating With Other Computer Programs

There may be occasions on which you wish to move data between PROPHET and another program on the NIH computer, or between the NIH computer and another computer. For example, you may already have data on another computer that you wish to use within PROPHET. You might also wish to run a program written in another language utilizing data stored within PROPHET. This might be an already-existing program that you do not wish to translate into PL/PROPHET; or one that does a lot of computation, which can be carried out more efficiently in a language like Fortran.

Whether you wish to communicate with another program on the NIH computer or with another computer altogether, it is necessary to pass data back and forth. The standard technique for accomplishing this is to write the PROPHET data into a disk file that the other program or computer can read. In order for PROPHET to have access to data from another program or another computer, the data must be written on the disk and it must be written using format conventions that PROPHET can understand. In order for another program to have access to PROPHET data, the data must be on the disk and it must be written using format conventions the program can understand. We discuss briefly below how to run an extra-PROPHET program, and some of the considerations involved in transferring data between computers. The bulk of this section, however, is devoted to a description of the PROPHET statements used to read or write data on the disk.

A. Communicating with Other Programs from PROPHET

Consider a hypothetical program named APRO, written in another language that you wish to utilize operating on PROPHET data and returning its results to PROPHET. You must be familiar with the format conventions of that language. You can then go through seven basic steps:

1) write your PROPHET data into a disk file in a format acceptable to APRO;
2) start up APRO;
3) read the data from the disk file into APRO;
4) proceed with the computations of APRO;
5) write the results from APRO into a disk file in a format acceptable to PROPHET;
6) return to PROPHET;
7) read the results from the disk file back into PROPHET.

Steps 3, 4, and 5 are accomplished within the extra-PROPHET program, APRO, and the details of those steps will depend on the language of the program. Steps 1 and 7 are carried out within PROPHET, using the input and output commands discussed later.

Running
Another
Program

To carry out step 2, the RUN command may be used. This takes the form

```
RUN file-name
```

The file named must contain a compiled, loaded, and saved version of your program. You should speak with your PROPHET representative to find out how to prepare your program (APRO) for running; the details of this will depend on the language in which your program is written.

To carry out step 6, there is a system program available which will return you to PROPHET when execution of your program is finished. This is called from the monitor (where your program will leave you after execution) by using the statement

```
CALL RUNRTN
```

Although it is possible to use the statement CALL PROPHET, calling RUNRTN returns you to the exact state you were in within PROPHET when the RUN command was given. All your variables will still be present. If you used the RUN command from within a PL/PROPHET procedure, it is essential to return to PROPHET with RUNRTN in order to continue execution of the procedure. (It is usually possible to call RUNRTN from within your program (APRO), as well. Call your PROPHET representative for information, since the details of this process again depend on the language of your program.)

B. Communicating with Other Computers

If your program APRO were on another computer, it would still be necessary to execute the steps outlined above, in order to transfer data from PROPHET to APRO and back again. In addition, there would be two steps between 1 and 2 to transfer the data from the NIH computer to the second computer and two steps between steps 6 and 7 to transfer the data back:

1) write your PROPHET data into a disk file in a format acceptable to APRO;
    1.1) copy the data from the disk file to magnetic tape, using a system program;
    1.2) copy the data from the tape to the other computer, using a system program;
2) start up APRO;
3) read the data into APRO;
4) execute the computations making up APRO;
5) write the results from APRO in a format acceptable to PROPHET;
    5.1) copy the data from the other computer to magnetic tape, using a system program;
    5.2) copy the data from the tape to the disk of the NIH computer, using a system program;
6) call PROPHET;
7) read the results from the disk file back into PROPHET.

All computers have system programs which can be used for copying data from one physical device to another, e.g. disk to tape. Computer center personnel will assist you in using these programs. In many cases, it will be possible to read data from the tape directly into your program APRO, thus combining steps 1.2 and 2. If a data file already exists on another computer that you want to use within PROPHET, steps 5, 6, and 7 may be used by themselves.

When you wish to transfer data between computers, the first step should always be to talk with personnel at the target site. They will inform you what physical means is most acceptable for data; e.g.e.g. DECtape or magnetic tape. They will also tell you of any restrictions in the format of data on the physical device. Since requirements vary from one installation to another, it is not possible to give a prescription here for accomplishing the data transfer process. The remainder of this section discusses the ways to accomplish steps 1 and 7. An overall example is given at the end of the section.

C. Using Disk Files

The primary device used to communicate data in the NIH computer is the disk. The disk is a data storage area of the computer which can be accessed not only by PROPHET, but by any program running on the computer.

1. Naming Disk Files

Data on the disk of the NIH computer is collected together in a file. This file is a series of characters in an area of the disk and must have a logical name by which it is referenced. File names must be of a particular format:

```
name[.extension]
```

The name field must contain six or fewer characters, and only alphabetic characters plus the ten digits may be used. The extension field is optional, and when present may contain three or fewer characters. Both name and extension should begin with a letter. These rules are slightly more restrictive than absolutely necessary for the PROPHET computer, but assure maximum compatibility with other language conventions. The following represent examples of legal file names.

```
MYPRC.PRO
PRO12
PRO1
PRCC.P1
```

A unique file name should be used for each different data file. When an already-existing file name is used, data will be written over what is already there. The same name is used to read a file as was used to create it.

It is possible to refer to a file created by another user. This is accomplished by appending to the file name the project-programmer number (PPN) of the file's owner inclosed in square brackets, for example:

```
file-name.extension[PPN]
```

Thus if you wanted to refer to a file named "MYPRO.DAT" which belonged to user [1200,1217], you would refer to the file as

```
MYPRO.DAT[1200,1217]
```

If you are writing a procedure which will be used by others, you should include your PPN, enclosed in square brackets, after the name of any file to be run using the RUN command. Usually you want data files to be read and written by the owner of the data files, so this feature would not be used in naming data files. In order for one user to access a file created by another, the protection codes on the file must be set to allow this. If you are sharing within your project, the protection will usually be correct. To share a file with someone outside your project, the protection will usually need to be changed. Talk with your system representative for further information on this point.

2. Reading and Writing Disk Files

Within PROPHET, input and output statements ordinarily refer to your terminal unless some other device is specified. You can input the contents of an entire disk file at one time, using the function

```
text-variable = %GETFILE(file-spec)
```

where "file-spec" is a variable or constant of type TEXT containing the name of the file you want to read, in the form

```
file-name.extension[PPN]
```

Note that although the PPN is optional, if used it must be enclosed in square brackets. %GETFILE returns a TEXT, which contains the contents of the entire file. For example, the statement

```
TERMUS=%GETFILE('FRMUS.PRO')
```

(where TERMUS has been declared a variable of type TEXT) will input the entire file FRMUS.PRO, as a single text, into the variable named TERMUS.

You can output an entire text into a disk file by using the procedure

```
%PUTFILE(file-spec,text-variable)
```

where "file-spec" is a variable or constant of type TEXT which contains the name of the file (in the same form as for %GETFILE) and "text-variable" is a TEXT variable or constant containing the text to be written into the file. For example, the statement

```
CALL %PUTFILE('NORMUS.PRO',DORMUS)
```

will output the contents of the TEXT variable DORMUS to the disk file NORMUS.PRO. If the file does not already exist on the disk, it will be created. If it does already exist, whatever is in it will be written over.

Opening and Closing Disk Files

In order to write or read less than an entire file onto or from the disk, you must first "open" the disk file. The statement to accomplish this for reading data from a disk file is

```
channel = INFILE file-spec
```

where "channel" is the name of a FIXED variable, and "file-spec" is a TEXT containing the name of the disk file to be read (it should be in the format described above for %PUTFILE and %GETFILE). It is never necessary to know the value of "channel"; only its name is used in the input statements. Each input statement reads data from the file sequentially. To "close" the input file, when all input operations relating to that file are completed, the statement

```
        CLOSEIN channel
```

is used, where "channel" is the variable named in the INFILE statement used to open the file. The CLOSEIN statement should be used when you are through reading from that file.

Analogous statements are used to open and close an output file. To open a file for output,

```
        channel = OUTFILE file-spec
```

is used. To close the output file, the statement

```
        CLOSEOUT channel
```

is used. The CLOSEOUT statement must be used before you leave PROPHET, or data written on that file will be lost.

It is possible to determine whether or not a file, either input or output, has already been closed, by use of the function

```
        close-flag = TESTCLOSE(channel)
```

where "channel" is the variable named in the INFILE or OUTFILE statement. TESTCLOSE is a function that returns a boolean value which will be TRUE if the file has been closed and FALSE if the file has not been closed. This is useful in a program since reading to the end of an input file automatically closes the file.

Input and Output Statements

There are a large number of different input and output statements available within PROPHET. Only two of them are suitable for use with formatted data. To read data into PROPHET, the statement

```
GETLINE(input-text,channel)
```

is used. "Channel" is the FIXED variable obtained in the INFILE statement. "Input-text" is the name of a variable previously declared to be of type TEXT. When the GETLINE statement is executed, one line of data in the file is read into the TEXT variable named. (Commands for breaking a line of text down into pieces are described in succeeding paragraphs.) To write data from PROPHET, the statement

```
PUTLINE(output-text,channel)
```

is used. "Channel" is the FIXED variable obtained using the OUTFILE statement. "Output-text" is the name of a TEXT variable or constant containing the data to be written into the file. PUTLINE writes one line of data, just as GETLINE reads one line. Any data written using PUTLINE can be read back in using GETLINE. The maximum number of characters that can be read using the GETLINE statement is 2000. Any characters beyond that before the end of the line will be read in with the next GETLINE statement. (One line of data also corresponds to one card image. Therefore, if you are interested in writing card images, you should not write more than 80 characters per line, i.e. per PUTLINE statement.) A line can be terminated by the occurrence of a carriage return-line feed character which is produced by the use of the RETURN key on your terminal. Note that the line terminator is discarded by GETLINE when a line is read. If two lines, each the result of a GETLINE, are concatenated, there will be no carriage return between them, i.e. they will appear as one line.

Example

Suppose you wish to write out some text onto a disk file which you name MDAT. This could be accomplished using the following statements:

```
DECLARE F1 FIXED;
F1=OUTFILE 'MDAT';
PUTLINE('THIS IS A TEST TEXT',F1);
CLOSEOUT F1;
```

To read the same data back in at another time, you must have a TEXT variable available to put the data into. The following statements could be used:

```
DECLARE JJ FIXED;
DECLARE VAR1 TEXT;
JJ=INFILE 'MDAT';
GETLINE(VAR1,JJ);
CLOSEIN JJ;
```

VAR1 now contains 'THIS IS A TEST TEXT'. Note that it is never necessary to examine the value of either F1 or JJ; only their variable names are actually used in the input or output statements.

D. Using Formatted Data

The PUTLINE and GETLINE statements only operate with data of type TEXT. In order to read and write numbers, the numerical values must be converted to a text representation. There is a great difference between the number 1.2 and the TEXT '1.2'. For this reason, functions are provided to encode and decode numerical data, i.e. to convert numbers to TEXT. These functions are not the same as the CTEXT function described in Section III above, since they put each number in a specified format. You must first use the DECLARE statement to create an empty variable of type TEXT.

1. Encoding and Decoding Data

To encode the data in a single variable, you use the function

```
encoded-text = ENCODE(variable-name,format-text)
```

The ENCODE function returns a TEXT representation of "variable-name" formatted according to the specifications supplied by "format-text". (The various kinds of format specifications are described in greater detail below.) "Format-text" is a TEXT variable or constant containing the actual format to be used for the encoding of the data.

You can build up texts for output through the PUTLINE statement by using the TEXT concatenation operator CAT. For example, if you had two variables to be encoded in the same text, you might use the statement

```
TT = ENCODE(VAR1,FOR1) CAT ENCODE(VAR2,FOR2)
```

Here VAR1 and VAR2 are the names of the variables containing the data to be written out, while FOR1 and FOR2 are the names of TEXT variables containing the formats to be used for each piece of data. TT is the name of the TEXT which is to be written out with the PUTLINE statement.

2. Decoding Data on Input

Data is decoded on input by performing operations on the TEXT variable read in with the GETLINE statement. The variable whose value is to be obtained from the input text must have already been declared to be of the proper type. When this is done, the following statement may be used to extract the data from the input text

```
variable-name = DECODE(text-variable,format-text)
```

Here "variable-name" is the name of the variable whose value is being obtained. "Text-variable" is the name of the TEXT input with the GETLINE statement. "Format-text" is the name of a TEXT variable containing the format to be used in decoding the input TEXT, or it may be the actual format enclosed in single quotation marks. Although the input TEXT may contain more than one data value in encoded form, each use of the DECODE function will extract only one value from that text. (For more complicated uses of input and output, it is always possible to use any of the TEXT operations described in Section III. above to break down the input TEXT variable. DECODE is a simple, straightforward function which is designed to be compatible with data formats of other languages.)

3. Formatting Numbers for Output

There are two types of format-texts that can be used for numbers, the "F" and "E" formats. The "F" format is used for output of ordinary numbers, the "E" format is used for exponential or scientific notation.

The "F" Format

The "F" format takes the form

F(total-width[,number-of-decimal-digits])

"Total-width" is an integer designating the total number of characters to be written out. The optional field "number-of-decimal-digits", tells how many digits are to follow the decimal place. If this field is omitted, the number is written out as an integer with no decimal point. "Total-width" must be large enough to accommodate the number being output, including the decimal point and sign, if any. When digits after the decimal place are included, the least significant may be left out by choosing a small value for the second field. Examples will illustrate the use of this format.

VALUE	FORMAT	OUTPUT
123456.78	F(15)	_________123457
123.	F(7,2)	_123.00
1234.	F(5,0)	1234.
-1234.	F(5,0)	error - width too small
123456.789	F(15,2)	______123456.79
123	F(4,0)	123.
1234.	F(6,2)	width too small for 1234.00

Note that unused places in "total-width" are filled in, from the left, by blanks (indicated in the above examples with the underscore character ("_"). The "number-of-decimal-places" is used to make up the number; the result must then fit in "total-width". Unused places in the digits after the decimal place are filled in with zeroes. This will only happen if the original number was exact. Round-off error within the computer may cause additional least significant figures to be output, especially if the number is a result of a computation.

The "E" Format

The "E" format is used to produce numbers in exponential form. This is particularly useful for very large or very small numbers. The "E" format has the form

E(total-width[,number-of-decimal-digits])

The two fields have the same meaning as for the "F" format. When the second field is omitted, the number to be output is always adjusted so that there is but one digit to the left of the decimal point, and the exponent is adjusted accordingly. "Total-width" must always be at least seven greater than the number of digits to follow the decimal point to allow room for the sign, decimal point, and exponent. In addition, extra zeroes will be added to the end of the number so that it always completely fills the specified field. Some examples will make this clear.

VALUE	FORMAT	OUTPUT
123456.78	E(11)	1.2346E+05
123.	E(12,2)	1230.00E-01
123.456	E(7,2)	error - width too small
123456.78	E(14,5)	123.45678E+03

In using the "E" format, you can always output an entire computer word using E(16).

4. Formatting TEXT for Output: the "A" Format

TEXT may be output using the "A" format, which takes the form

A(total-width)

Here only "total-width" is needed. If the actual text to be output is smaller than the field specified, the characters are left-adjusted in the field and the remainder of the field is filled out with blanks. If the actual text to be output is larger than the field specified, the left-most characters are taken. Some examples may make this clear.

VALUE	FORMAT	OUTPUT
ABC	A(10)	ABC_______
GRAPEFRUIT	A(5)	GRAPE
____ABC	A(5)	____A

It is not actually necessary to use the "A" format to output TEXT since the contents of the text string may be concatenated directly into the output text. For example, suppose you had a number named VAR1 to output and a label to go with it in TXT1. These could both be placed in the output TEXT variable, TT, by means of the following statement

```
TT = ENCODE(VAR1,FOR1) CAT TXT1
```

This way of formatting TEXT will neither truncate nor pad the original text. It is also possible to associate a small text directly with the format-text. For example, the format-text for a number might have the value

```
FOR1 = 'F(8,2)" MIN."'
```

which would result in a text field 5 characters long being output right after the number. Note that the TEXT, " MIN.", is enclosed in <u>double</u> quotation marks in this usage. Suppose you have the following variables with the values shown:

VAR1	equal to	25
TXT1	equal to	' MIN.'
FOR1	equal to	'F(8,2)'
FOR2	equal to	'F(8,2)" MIN."'
FOR3	equal to	'A(5)'

Then the following three statements would all produce the identical output TEXT, viz. TT would be equal to '___25.00_MIN.'

```
TT = ENCODE(VAR1,FOR1) CAT ENCODE(TXT1,FOR3)

TT = ENCODE(VAR1,'F(8,2)') CAT TXT1

TT = ENCODE(VAR1,FOR2)
```

5. Formatting on Input

When numbers are being input, the same two types of format-texts are available as for output, viz., the "F" and "E" formats. In general, when formatting numbers, the format-text should match the format of the number in the input text. If "total-width" is the proper size, however, decimal points in the input numbers will override the decimal points specification of the format-text. When using the "F" format, blanks may precede the number without causing any difficulty. Blanks may follow the number and be interpreted as zeroes if they fall inside the specified field (see the examples below). This is only true for "F" format; in "E" format, an error will occur, as no embedded blanks are allowed in "E" formats. If the number of digits following the decimal point is greater than that specified, the number will be rounded and truncated to fit. The following examples show how format-texts may be used in input.

INPUT TEXT	FORMAT	VALUE RETURNED
_123.__	F(7,2)	123.00
_123	F(4,0)	123.
123_	F(4,0)	123.
123	F(3)	123
____	F(4,0)	error-no data
.00012345	F(8,6)	.000123
1.234E+05	E(10,3)	1.234E+05

(In these examples, blank spaces in the input text are represented with the underline character "_"). Note that an entirely empty field, i.e. one all blank, will not return the value zero, but will cause an error.

Formatting Text on Input

When text is being input, the "A" format is used in DECODE to read it in. Exactly the number of characters specified will be extracted up to the total number of characters in the line, regardless of what characters are in the input text.

Locating a Value in the Input Text

When data is being decoded from an input text, it is necessary to know where in the text the value is located. The format-texts described in the preceding paragraphs assume that the value is located at the beginning of the input text. It is, therefore, necessary to be able to skip over a value that has already been extracted from the input.

The "X" Format

This is accomplished with the "X" format, which takes the form

```
X(skip-width)
```

Use of this format-text will cause DECODE to skip the number of characters specified by "skip-width". The "X" format is used in conjuction with one of the other types of format-texts in a single DECODE statement. For example, suppose you had an input text which contained the following values:

```
___FIRST_PAIR_____123.4568____1.2345678E+02___
```

The following statements would place the three pieces of data in the variables TXT1, VAR1, and VAR2.

```
TXT1 = DECODE(TT,'A(13)')
VAR1 = DECODE(TT,'X(17)F(9,4)')
VAR2 = DECODE(TT,'X(29)E(14)')
```

Note that the value of the format-text may be placed directly into the argument list of DECODE. In this example, blank spaces in the input text are again indicated by underscores. Note that the TEXT variable, TXT1, contains three leading spaces as well as the characters "FIRST_PAIR". Other format-texts besides those given in the example could be used to decode the input text, TT.

E. An Example of Extra-PROPHET Communication and File-Handling

In order to illustrate the use of some of the procedures described in this chapter, an example is given below. This example assumes that you have a fast matrix inversion program, called MINV, written in Fortran, which you wish to utilize with data you already have in PROPHET. Shown here is a procedure which can be called to invert a PROPHET table. Included are the statements needed to write your data on the disk for use by MINV, the statement which calls MINV, and the statements needed to read the resulting inverted matrix back into a PROPHET table. The procedure assumes that RUNRTN has been called either from within MINV or that you called it manually from the monitor when MINV has finished execution.

```
INVERTTABLE: PROCEDURE(TO);

/* Declare all variables to be used */

    DECLARE TO TABLE;
    DECLARE (NUMROWS,NUMCOLS,I,OUTNO,J,INNO,K) FIXED;
    DECLARE ACOL(*) FIXED;
    DECLARE (TODAT,TIDAT,IDENT)(*,*) FLOAT;
    DECLARE TXT TEXT;

    NUMROWS = %LASTROW(TO);
    NUMCOLS = %LASTCOL(TO);

/* Check table for invertability */

    IF NUMROWS <> NUMCOLS THEN DO;
        TYPE 'CAN ONLY INVERT SQUARE TABLES';
        RETURN;
    END;

    IF NUMROWS = 0 THEN DO;
        TYPE 'CANNOT INVERT AN EMPTY TABLE';
        RETURN;
    END;

/* Set up column specification array */

    DO I = 1 TO NUMCOLS;
        ACOL(I) = I;
    END;

/* Write table contents into an array */

    CALL %T2A(TO,ACOL,1,NUMROWS,TODAT,'');

/* Output the array to the disk, one value per line */

    OUTNO = OUTFILE 'MAT.DAT';

    TXT = ENCODE(NUMROWS,'F(10)') CAT ENCODE(NUMCOLS,'F(10)');
    PUTLINE(TXT,OUTNO);


    DO I = 1 TO NUMROWS;
        DO J = 1 TO NUMCOLS;
            TXT = ENCODE(TODAT(I,J),'E(16)');
            PUTLINE(TXT,OUTNO);
        END;
    END;

    CLOSEOUT OUTNO;
```

```
/* Execute the Fortran program MINV */
/* MINV reads the data in 'MAT.DAT', does the
   inversion, writes the inverse matrix into 'INV.DAT',
   and calls RUNRTN */

    RUN 'MINV';

/* Clean up the screen on returning */

    CLEAN;

/* Input the results from MINV */

    INNO = INFILE 'INV.DAT';

    DO I = 1 TO NUMROWS;
        DO J = 1 TO NUMCOLS;
            GETLINE(TXT,INNO);
            TIDAT(I,J) = DECODE(TXT,'E(16)');
        END;
    END;

    CLOSEIN INNO;

/* Write inverse array into table */

    CALL %A2T(TO,ACOL,1,NUMROWS,TIDAT,'');
    RETURN;
END;
```

XII. Vector, Matrix and Transformation Procedures and Functions

by
Mark A. Lavin
Bolt Beranek and Newman Inc.

One of the major features of the PL/PROPHET language is that it allows manipulation of high-level data types (e.g., TABLEs, MOLECULEs) as well as numbers and character strings. Another example of this feature is the facility for representing and manipulating vectors and matrices. The vector and matrix facilities were originally designed to support the geometric computations inherent in describing and displaying molecules, but have a broad range of other applications. The vector and matrix data types and operations have several advantages:

(1) Readability: manipulations involving vectors and matrices are written in shorter and more natural notation.

(2) Efficiency: the vector and matrix operators are considerably faster and more storage-efficient than equivalent routines written in PL/PROPHET.

The remainder of this chapter is divided into seven sections: Section A describes the representation of vectors and matrices as PL/PROPHET data-types and introduces some notation conventions used in the remainder of the chapter. Sections B and C describe the operators and procedures used for manipulation of vectors and matrices. Section D introduces the concept of the transformer, a matrix used for three-dimensional transformations, and describes operators and procedures used with transformers. Section E discusses some miscellaneous routines, and Section F provides some examples of how vectors, matrices, and transformers can be used. Finally, Section G contains some details about the implementation of transformers.

A. Representation of Vectors and Matrices

For purposes of this discussion, vectors are defined to be N-tuples (1-dimensional arrays) of real numbers, and matrices are defined to be N-row by M-column rectangular arrays of real numbers. These two data types are represented in PL/PROPHET by the data types FLOAT array and FLOAT array array respectively. In fact, the system allows the interchangable use of the equivalent types, i.e.,

```
DECLARE V VECTOR;
DECLARE M MATRIX;
```

is equivalent to:

```
DECLARE V FLOAT ARRAY;
DECLARE M FLOAT ARRAY ARRAY;
```

or to:

```
DECLARE (V(*), M(*,*)) FLOAT;
```

Vectors and matrices may be used as top-level variables, as local variables, and as arguments to and results from procedures. They may be constructed and their components may be accessed by the array handling mechanisms of PL/PROPHET, e.g.:

```
V = [1., 0., 0.];            /* ARRAY CONSTANT */

M(1,1) = 1.0;  M(1,2) = 0.0;     /* SUBSCRIPTING */
M(2,1) = 0.0;  M(2,2) = V(1);
```

In the latter cases the first dimension (subscript) of a matrix corresponds to a row number, while the second corresponds to a column number.

Pitfalls

There are several mistakes to avoid when working with VECTORs and MATRIXes in PL/PROPHET:

(1) VECTOR is the equivalent of (*) FLOAT, not FLOAT(3). Thus, when we speak of an N-vector, it is actually a "variable length array whose TOP = N." The same facts pertain to MATRIXes.

(2) A scalar (FLOAT) is not the same thing as a 1-vector, which is not the same thing as a 1x1-matrix,

even though all might be TYPEd out identically by PROPHET.

(3) The temptation to build up vectors and matrices using array constants must be resisted if any elements are variable; e.g.,

```
X = 13.;  Y = 3 * X**2;  Z = Y - 2.*X;

V = [X, Y, Z];
```

will not have the desired effect. Instead, one must say:

```
X = 13.;  Y = 3 * X**2;  Z = Y - 2*X;
V(1) = X;  V(2) = Y;  V(3) = Z;
```

Notation Conventions

The descriptions below use a mixture of mathematical and computer notation:

element:	One of the numbers in a matrix or vector
N-vector:	A vector with N elements
MxN Matrix:	A matrix with M rows and N columns
size:	The number of elements in a vector The number of rows, columns in a matrix
Vi:	The i-th element of vector V
Mi,j:	Element of matrix M in row i, column j

B. Algebraic Operators

Many of the operations on vectors and matrices are provided in the form of arithmetic operators (see Table 12.1). Some of these (e.g., +, -) are simply extensions of the operators for FIXED and FLOAT numbers, while others (e.g., DOT, CROSS) apply only to vectors and matrices.

The Addition Operator (+)

The expression VA + VB represents the vector sum of two vectors VA and VB, that is, a vector VC in which VCi = VAi + VBi. Similarly, the expression MA + MB is a matrix MC in which MCi,j = MAi,j + MBi,j. In both cases, the sizes of the operands must agree, i.e., VA and VB must both be N-vectors, and MA and MB must both be MxN matrices; the size of the result agrees with the size of the operands.

The Subtraction Operator (-)

The expression VA - VB represents the difference of the vectors VA and VB, that is, a vector VC in which VCi = VAi - VBi. The expression MA - MB yields another matrix MC in which MCi,j = MAi,j - MBi,j. As with addition, the sizes of the operands and the result must agree.

The Negation Operator (-)

The expression -V is the negation of the vector V, that is, a vector whose elements are the negative of the components of V. Similarly, the expression -M represents the element by element negation of the matrix M.

The Multiplication Operator (*)

The multiplication operator * has a number of different interpretations, depending on the data-type of its two operands:

(1) VA * VB (where the sizes of VA and VB agree) results in a scalar (FLOAT) which is the inner, or dot product of VA and VB:

$$VA * VB = \sum_{i} VAi * VBi$$

TABLE 12.1: VECTOR and MATRIX Operations

FORM	TYPE OF RESULT	DESCRIPTION OF RESULT
V1 + V2	VECTOR	Vector Addition; sizes must agree
M1 + M2	MATRIX	Matrix Addition; sizes must agree
- V	VECTOR	Negation of a vector
- M	MATRIX	Negation of a matrix
V1 - V2	VECTOR	Vector subtract; sizes must agree
M1 - M2	MATRIX	Matrix subtract; sizes must agree
F * V V * F	VECTOR	Scalar multiplication of vector
F * M M * F	MATRIX	Scalar multiplication of matrix
V * M	VECTOR (ROW)	Pre-multiplication of matrix by row vector; sizes must agree
M * V	VECTOR (COLUMN)	Post-multiplication of matrix by column vector; sizes must agree
M1 * M2	MATRIX	Matrix multiplication; M1 is NxM M2 is MxQ, result is NxQ
V1 DOT V2	FLOAT	Dot (inner) product of vectors; sizes must agree
V1 CROSS V2 (2-vectors)	FLOAT	Cross product of 2-vectors
V1 CROSS V2 (3-vectors)	VECTOR	Cross product of 3-vector; result is 3-vector

Notation conventions:

V,V1,V2	Vectors
M,M1,M2	Matrices
F	Scalar (FLOAT)

(2) The expression VA * MB (where VA is an N-vector and MB is an NxM matrix) results in an M-vector VC which is the product of premultiplying MB by the (row) vector VA:

$$VC_j = \sum_{i=1}^{N} VA_i * MB_{i,j}$$

(3) The expression MA * VB (where MA is an NxM matrix and VB is an M-vector) results in an N-vector VC which is the product of postmultiplying MB by the (column) vector VB:

$$VC_i = \sum_{j=1}^{M} MA_{i,j} * VB_j$$

(4) The expression MA * MB (where MA is an NxM matrix and VB is an MxQ matrix) results in an NxQ matrix product MC:

$$MC_{i,j} = \sum_{k=1}^{M} MA_{i,k} * MB_{k,j}$$

(5) The expressions F * V and V * F are the scalar product of the vector V and the FLOAT number F, that is, a vector VB such that $VB_i = F * V_i$. Similarly, the expressions F * M and M * F are the scalar product of the matrix M and the FLOAT number F, that is, a matrix MB such that $MB_{i,j} = F * M_{i,j}$.

The DOT Operator

The expression VA DOT VB represents the inner (dot) product of the vectors VA and VB (whose sizes must agree), that is, a number:

$$VA \text{ DOT } VB = \sum_{i} VA_i * VB_i$$

The expressions VA DOT VB and VA * VB are equivalent, but the former may be more indicative of the intent.

The CROSS Operator

The result of the CROSS operator depends on the size of the vectors to which it is applied. If VA and VB are 2-vectors, then VA CROSS VB yields a FLOAT number:

```
VA(1) * VB(2) - VB(1) * VA(2)
```

If VA and VB are both 3-vectors, then VA CROSS VB yields a 3-vector VC which is the vector product of VA and VB:

```
VC(1) = VA(2)*VB(3) - VA(3)*VB(2);
VC(2) = VA(3)*VB(1) - VA(1)*VB(3);
VC(3) = VA(1)*VB(2) - VA(2)*VB(1);
```

The expression VA CROSS VB is undefined if VA and VB are not both 2-vectors or 3-vectors.

Precedence of Operators

The precedences (order of evaluation) for the operators +, -, and * *are the same* for vectors and matrices as for FLOATs and FIXEDs (see Chapter I.D above). The precedences for DOT and CROSS are the same as for vector multiplication; all operators are left-associative, e.g.,

```
VA * VB * VC  is equivalent to  (VA * VB) * VC
```

Note that the * operator, when used with a matrix operand, is emphatically not commutative; that is, in general, MA*MB <> MB*MA.

best copy available

C. Vector and Matrix Functions

Other vector and matrix operations are implemented as function procedures (see Table 12.2), that is, procedures which are called with one or more vector and/or matrix arguments, and which return a scalar, vector or matrix result.

MAG,
%MAGSQD

The expression MAG(V) returns the scalar magnitude, or length of the vector V, that is, a FLOAT number such that:

$$\text{MAG(V)} = \sqrt{\sum_{i=1}^{N} Vi * Vi}$$

The expression %MAGSQD(V) returns the square of the magnitude of V; it is much faster than MAG, since it does not extract the square root.

%UNITVEC

The expression %UNITVEC(V) returns a unit VECTOR pointing in the same direction as the vector V, that is a vector whose magnitude is 1.0 and whose elements are proportional to those of V. An important exception occurs if the magnitude of V is zero: in that case, %UNITVEC(V) is a vector the same size as V with all 0.0 elements (effectively, a copy of V).

%PERPTO

The expression %PERPTO(V), where V is a N-vector, returns another N-Vector VP such that V and VP are perpendicular (V DOT VP = 0.0) and VP is non-zero (MAG(VP) > 0.0). VP is only guaranteed to have these two properties, and is otherwise somewhat randomly chosen. One exception can occur: if V is a non-zero 1-vector, the vector [0.] will be returned, violating the "non-zero" property.

%TRANSPOSE

The expression %TRANSPOSE(M) returns a MATRIX which is the transpose of the matrix M; that is, if M is an PxQ matrix, then %TRANSPOSE(M) returns a QxP matrix MT such that MTi,j = Mj,i.

%INVERSE

The expression %INVERSE(M) returns a MATRIX which is the multiplicative inverse of the matrix M; that is, if M is an NxN matrix, then %INVERSE(M) will return an NxN matrix MI such that M * MI = MI * M = NxN identity matrix (1's on the diagonal, 0's elsewhere). Several exceptions can occur: if M is not a square

TABLE 12.2: VECTOR and MATRIX Functions

FORM	TYPE OF RESULT	DESCRIPTION OF OPERATION
MAG(V)	FLOAT	Magnitude of vector
%MAGSQD(V)	FLOAT	Square of magnitude of vector; (faster than MAG)
%UNITVEC(V)	VECTOR	Unit vector along V
%PERPTO(V)	VECTOR	Vector perpendicular to V
%TRANSPOSE(M)	MATRIX	Transpose of matrix
%INVERSE(M)	MATRIX	Inverse of (square) matrix
%DET(M)	FLOAT	Determinant of (square) matrix

Notation conventions:

V,V1,V2	Vectors
M,M1,M2	Matrices
F	FLOAT (scalar)

matrix, %INVERSE will ABORT. If M is non-invertible (determinant = 0), then the result returned will be an EMPTY matrix to signal this condition.

%DET The expression %DET(M), where M is an NxN (square) matrix, returns the determinant of M.

D. Transformers

Definition of Transformers

One of the major uses of vectors and matrices within the PROPHET system procedures is in computations involving three-dimensional (3D) geometry. A 3-vector [X Y Z] can represent a point in 3D space with cartesian coordinates X, Y, and Z. The vector [X Y Z] can also represent the direction of a set of lines in 3D space, namely all lines parallel to one drawn from the origin to the point [X Y Z].

Often, in geometric computations, one wishes to transform points in 3D space by shifting them (translation), by rotating them about some axis, by shrinking or expanding space about some point (scaling), or by combinations of these transformations. PL/PROPHET provides a convenient and efficient way to transform points in 3D space: a transformer is a MATRIX which describes a particular transformation in 3D space (Section G describes the details of implementation). PL/PROPHET provides two sets of tools for working with transformers: first, a means of applying transformers to points (3-vectors) to transform them in 3D space, and second, a means of generating and combining transformers to perform particular kinds of tranformations.

The XFM Operator

The expression V XFM X, where V is a 3-vector and X is a transformer (MATRIX), yields another 3-vector having the following interpretation: if V represents a point in 3D space, and X represents a transformation, then V XFM X represents the new point which results from performing the specified transformation (movement) on the specified point. For example, if X = %TRANSLATION(3.,4.,5.) (see below), then V XFM X is a point which results from moving point V by 3 units in the X-direction, 4 units in the Y-direction, and 5 units in the Z-direction.

Combining Transformers

Sometimes one wishes to apply several tranformations to a point(s). One approach is to define several transformers and apply them in succession. For example, suppose X1 is a translation and X2 is a rotation; to achieve the effect of translating, then rotating a point V, one could write:

```
(V XFM X1) XFM X2
```

A different approach is to recognize that "translating, then rotating" is itself a transformation, which can be represented by a transformer. Thus, in PL/PROPHET, the expression X1 XFM X2 yields a <u>transformer</u> which has the same effect as the successive <u>application</u> (<u>composition</u>) of X1 followed by X2. That is,

```
(V XFM X1) XFM X2  =  V XFM (X1 XFM X2)
```

(thus, one may write V XFM X1 XFM X2 without regarding the order of evaluation).

In situations where a combination of transformers must be applied many times (e.g., to an array of points), it is often convenient to define a temporary variable to hold the combined transformer, thus:

```
/* XFMALL applies transformer X1, then transformer  X2  to
   an array of points PA, to yield another array TPA */

   XFMALL:  PROCEDURE(PA,X1,X2,TPA);

   DECLARE     /* ARGUMENTS */
        PA          VECTOR ARRAY,  /* Untransformed pts. */
       (X1,X2)      MATRIX,        /* Transformers */
        TPA         VECTOR ARRAY;  /* Transformed pts. */

   DECLARE     /* LOCALS */
       X1THENX2     MATRIX,     /* Combined trans. */
       I            FIXED;      /* Loop index */

       X1THENX2 = X1 XFM X2;    /* Combination */

       DO I = 1 TO TOP(PA);     /* Apply it to pts. */
           TPA(I) = PA(I) XFM X1THENX2;
       END;  /* DO I */

   END;  /* XFMALL */
```

Notice that in the cases above where transformers are associated with arguments (X1, X2) or local variables (X1THENX2), they are DECLAREd as MATRIXes.

Transformer-Building Procedures

To facilitate constructing transformers, a number of procedures return transformers to perform several classes of geometrically meaningful transformations in 3D space (see Table 12.3). Remember that transformers are of type MATRIX, and must be DECLAREd if they are to be stored in variables or passed as arguments.

%SCALE,
%SCALEV

The expression %SCALE(SX,SY,SZ), where SX, SY, and SZ are FLOATs, yields a transformer which has the effect of scaling 3D space by SX in the x-direction, SY in the y-direction, and SZ about the z-direction, about the origin. For example, if V is the vector [Vx Vy Vz], then V XFM %SCALE(SX,SY,SZ) yields the vector [SX*Vx SY*Vy SZ*Vz]. The function %SCALEV differs from %SCALE only it that it takes a single argument, which is a vector whose first three components are used as SX, SY, and SZ.

%SCALEABOUT

%SCALEABOUT(SX,SY,SZ,CX,CY,CZ) yields a transformer which scales by SX, SY, and SZ about the point [CX CY CZ]. This differs from %SCALE, which scales about the origin; that is,

```
%SCALE(SX,SY,SZ) = %SCALEABOUT(SX,SY,SZ,0.,0.,0.)
```

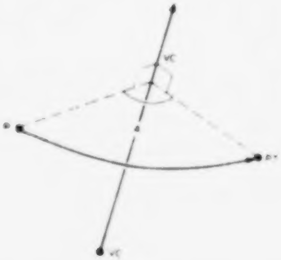
%TRANSLATION,
%TRANSLATEV

The expression %TRANSLATION(TX,TY,TZ) yields a tranformer which <u>translates</u> (shifts) points by TX in the x-direction, TY in the y-directions, and TZ in the z-direction. The procedure %TRANSLATEV takes a single argument, a vector whose first three elements specify the translations.

%ROTATION

The expression %ROTATION(VC,VA,A) yields a transformer which rotates points in 3D space A degrees around an axis having direction VA (a 3-vector), passing through point (3-vector) VC. For example,

```
PT = P XFM %ROTATION(VC,VA,A)
```



If VA is a zero-vector (MAG(VA) = 0.) then %ROTATION will ABORT.

TABLE 12.3: Transformer Operations

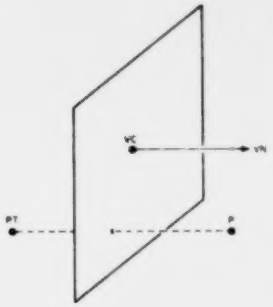
FORM	TYPE OF RESULT	DESCRIPTION OF RESULT
V XFM T	VECTOR	Transformation of vector
T1 XFM T2	TRANS-FORMER	Composition of transformations
%TRANSLATION(X,Y,X)	"	Transformer to translate by X,Y,Z
%TRANSLATEV(V)	"	Transformer to translate by V(1), V(2), V(3)
%ROTATION(VC,VA,A)	"	Transformer to rotate about axis vector VA, center VC, by A degrees
%SCALE(SX,SY,SZ)	"	Transformer to scale by factors SX, SY, and SZ about the origin
%SCALEV(V)	"	Transformer to scale by V(1), V(2), and V(3) about the origin
%SCALEABOUT (SX,SY,SZ,CX,CY,CZ)	"	Transformer to scale by SX, SY, SZ about center CX, CY, CZ
%REFLECTION(VC,VN)	"	Transformer to reflect through plane containing VC, normal VN
%INVXFM(T)	"	Inverse of transformer ("undoes T")

Notation conventions:

V,VA,VC	Vectors
T,T1,T2	Transformers (MATRIXes)
A,SX,SY,SZ	Scalars (FLOATs)
X,Y,Z	Scalars (FLOATs)
CX,CY,CZ	Scalars (FLOATs)

%REFLECTION The expression %REFLECTION(VC,VN) yields a transformer which performs a reflection of points through a mirror plane defined by the 3-vectors VC and VN: VC is a point contained in the mirror plane, and VN is the direction of the normal (perpendicular vector) to the plane. For example:

```
PT = P XFM %REFLECTION(VC,VN)
```



%INVXFM The expression %INVXFM(X) yields a transformer which is the inverse of the transformer X; that is, it "undoes" the effect of X. Thus, for all 3-vectors V:

```
V XFM (X XFM %INVXFM(X)) = V XFM (%INVXFM(X) XFM X) = V
```

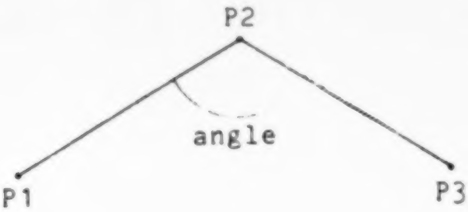
The procedure will ABORT if the argument is not a transformer (4x3 matrix), or if the transformation X is non-invertible. This can only arise if X is a scaling transformation (%SCALE, %SCALEV) in which one or more scale factors was 0., or if X was created by combining some other non-invertible transformer using XFM.

E. Miscellaneous Vector and Matrix Procedures

This section contains a grab-bag of functions for vectors, matrices, and transformers not mentioned in sections above (see Table 12.4).

%ANGLE3

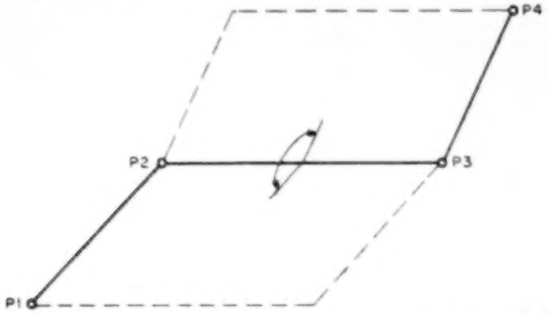
The function %ANGLE3(P1,P2,P3) returns a FLOAT which is the number of degrees (0. to 180.) in the angle formed by the (2- or 3D space) points P1, P2, and P3, thus:



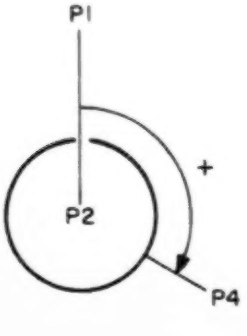
%ANGLE3 returns 0 degrees if P1 = P2 or P2 = P3 or P1 = P3.

%ANGLE4

The expression %ANGLE4(P1,P2,P3,P4) yields a FLOAT which is the number of degrees (-180. to 180.) in the torsion angle formed by the points (3-vectors) P1, P2, P3, and P4, that is, the angle between the planes defined by P1,P2,P3 and by P2,P3,P4, thus:



The sign convention is that used by stereochemists, as illustrated in the following Newman projections:



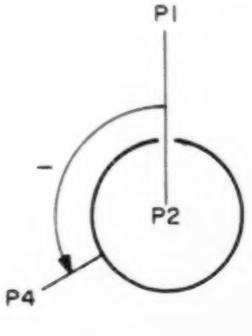


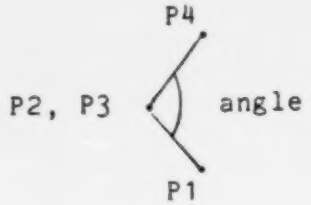
TABLE 12.4: Miscellaneous VECTOR and MATRIX Procedures

FORM	TYPE OF RESULT	DESCRIPTION OF RESULT
%ANGLE3(P1,P2,P3)	FLOAT	Angle (degrees) formed by three points (2D or 3D)
%ANGLE4(P1,P,P3,P4)	FLOAT	Torsion angle (degrees) formed by four points (3D)
%MATDIMEN(M,NR,NC)	---	NR, NC set to number of rows and cols in M, or EMPTY if M isn't rectang.
%IDENTITYMATRIX(N)	MATRIX	NxN identity matrix

Notation conventions:

P1,P2,P3,P4	Points
N,NR,NC	Fixed
M	Matrix

If P2 and P3 are coincident, the torsion angle is not well defined. In that case, the angle between the lines P2-P1 and P2-P4 is returned, thus:



If P1-P2-P3 are colinear, or if P2-P3-P4 are colinear, the torsion angle is undefined, but 0. degrees is returned.

%MATDIMEN

The procedure %MATDIMEN can be used to determine the size of a matrix through the statement:

```
CALL %MATDIMEN(M,NR,NC)
```

Where M is a MATRIX, and NR and NC are FIXEDs. Upon return, NR and NC will be set to the number of rows and columns in M. However, if M is not a rectangular matrix, then NR and NC will be set to EMPTY.

%IDENTITYMATRIX

The expression %IDENTITYMATRIX(N), where N is a FIXED, yields an NxN <u>identity matrix</u> of the form:

$$\begin{bmatrix} 1. & 0. & 0. & \dots & 0. \\ 0. & 1. & 0. & \dots & 0. \\ 0. & 0. & 1. & \dots & 0. \\ & \dots & & & \\ 0. & 0. & 0. & \dots & 1. \end{bmatrix}$$

An EMPTY matrix is returned if N < 1.

F. Examples of Vectors and Matrices

This section presents several examples of how the vector and matrix routines might be used:

Point
Spread

It is sometimes useful to have a measure of the spread of a set of points, whether they are in a plane or in 3D-space. One possible definition of the spread is suggested by analogy with the standard deviation of a set of scalars: the spread of a set of points is the root-mean-square distance of the points from the "mean point". The latter can also be defined by analogy to the mean of a set of scalar points. Thus:

$$\text{MEAN:} \quad \bar{V} = \frac{1}{N} \sum_{i=1}^{N} Vi$$

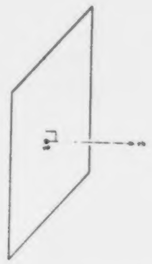
$$\text{SPREAD} = \frac{1}{N} \sqrt{\sum_{i=1}^{N} |\,(Vi - \bar{V})\,|^{2}}$$

The following program computes the spread of a set of points, expressed as an array of vectors. Notice that no mention of the size of the vectors is made, so the program can work in an arbitrary N-dimensional space.

```
/* PTSPRD RETURNS SPREAD OF A SET OF N-SPACE POINTS */

PTSPRD: PROCEDURE(PTS);
    RETURNS FLOAT;

DECLARE             /* ARGUMENTS */
    PTS     VECTOR ARRAY;   /* SET OF POINTS */

DECLARE             /* LOCALS */
    N       FIXED,      /* NO. OF PTS. */
    I       FIXED,      /* INDEX */
    SUMPT   VECTOR,     /* SUM OF PTS */
    AVGPT   VECTOR,     /* AVG. PT. */
    SUMD2   FLOAT;      /* SUM DISTS**2 */

    N = TOP(PTS);
    IF (N = 0) THEN RETURN(0.);

/* COMPUTE AVERAGE POINT */

    SUMPT = PTS(1);
    DO I = 2 TO N;  /* SUM UP PTS */
        SUMPT = SUMPT + PTS(I);
    END;  /* DO I */

    AVGPT = (1.0/CFLOAT(N)) * SUMPT;

/* COMPUTE SUM OF SQUARES OF DISTS. TO AVG. POINT */

    SUMD2 = 0.0;

    DO I = 1 TO N;
        SUMD2 = SUMD2 + %MAGSQD(PTS(I)-AVGPT);
    END;  /* DO I */

/* RETURN ROOT MEAN SQUARE */

    RETURN (SQRT(SUMD2/CFLOAT(N)));

END;  /* PTSPRD */
```

Planes and Things

Many geometrical computations involve descriptions of planes in 3D-space. One method of representing planes is by a pair of vectors, VP and VN, where the former is a point contained in the plane and the latter is a unit vector in direction perpendicular to the plane, thus:



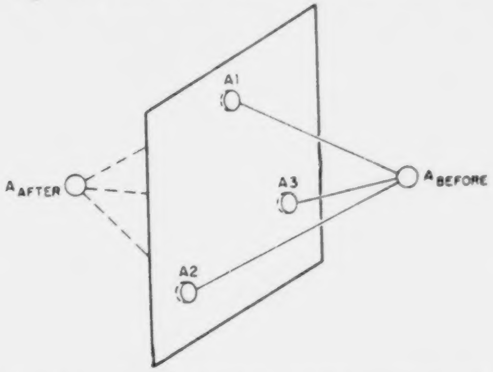
The following routines work with this VP,VN description of planes.

```
/* PLANENORM RETURNS A 3-VECTOR WHICH IS THE NORMAL TO THE
   PLANE FORMED BY THE THREE ARGUMENT POINTS (3-VECTOR) */

   PLANENORM:  PROCEDURE(P1,P2,P3);
       RETURNS VECTOR;

   DECLARE     /* ARGUMENTS */
       (P1,P2,P3) VECTOR;  /* 3 POINTS [IN] */

   DECLARE     /* LOCALS */
       VN  VECTOR;     /* NORMAL VEC. */

   /* COMPUTE UNIT NORMAL VECTOR; RETURN IF NON-ZERO,
      ELSE RETURN EMPTY */

       VN = %UNITVEC ((P2 - P1) CROSS (P3 - P2));
       IF (%MAGSQD(VN) = 0.0) THEN CLEAR VN;
       RETURN (VN);

   END;  /* PLANENORM */


/* PLANEDIST RETURNS THE (SIGNED) DISTANCE FROM A POINT TO
   A PLANE DESCRIBED BY TWO VECTORS */

   PLANEDIST: PROCEDURE(PT, PLP, PLN);
       RETURNS FLOAT;

   DECLARE     /* ARGUMENTS */
       PT  VECTOR,     /* THE POINT */
       PLP VECTOR,     /* PT. IN THE PLANE */
       PLN VECTOR;     /* UNIT NORMAL TO PLANE */

       RETURN ((PT-PLP) DOT PLN);  /* THAT'S ALL */

   END;  /* PLANEDIST */
```

```
/* PLANEPROJ RETURNS THE PROJECTION OF A POINT ON A  PLANE
   DESCRIBED BY TWO VECTORS */

   PLANEPROJ: PROCEDURE(PT, PLP, PLN);
       RETURNS VECTOR;

   DECLARE      /* ARGUMENTS */
       PT  VECTOR,     /* THE POINT */
       PLP VECTOR,     /* PT. IN PLANE */
       PLN VECTOR;     /* UNIT NORMAL TO PLANE */

   DECLARE      /* LOCALS */
       D   FLOAT;  /* (SIGNED) DIST. TO PLN */

       D = PLANEDIST (PT, PLP, PLN);

       RETURN (PT - D*PLN);

   END;  /* PLANEPROJ */
```

Reflecting a Molecule

This example presents the use of a transformer to alter the coordinates of a molecule. Specifically, it reflects the coordinates through a mirror plane which is defined by three of the atoms in the molecule, thus:



(Note that this procedure uses some molecule functions which are described in Chapter 9 above.)

```
/* REFLECTMOL REFLECTS THE STORED MODEL COORDINATES OF ITS
   ARGUMENT  MOLECULE  THROUGH A PLANE DETERMINED BY THREE
   ATOMS, WHOSE INDICES ARE THE 2ND  THRU  4TH  ARGUMENTS.
   ASSUMES   ALL  ATOMS  HAVE  COORDINATES  WHICH  CAN  BE
   RETRIEVED WITH %GETMVEC AND MODIFIED WITH %SETMVEC */

   REFLECTMOL:  PROCEDURE(MOLQ,A1,A2,A3);

   DECLARE          /* ARGUMENTS */
       MOLQ         MOLECULE,   /* REFLECTEE */
       (A1,A2,A3)   FIXED;      /* ATOM INDICES */

   DECLARE          /* LOCALS */
       (VM1,VM2,VM3)   VECTOR,     /* ATOM LOCATIONS */
       RFLXFM          MATRIX,     /* TRANSFORMER */
       I               FIXED,      /* LOOP INDEX */
       VM              VECTOR;     /* ATOM LOC */

    /* RETRIEVE LOCATIONS OF ATOMS DEFINING THE
       REFLECTION PLANE */

       VM1 = %GETMVEC(MOLQ,A1);
       VM2 = %GETMVEC(MOLQ,A2);
       VM3 = %GETMVEC(MOLQ,A3);

    /* COMPUTE NORMAL TO REFLECTION PLANE; IF EMPTY
       EXIT WITHOUT DOING ANYTHING */

       PLN = PLANENORM(VM1,VM2,VM3);
       IF (EMPTY(PLN)) THEN RETURN;

    /* COMPUTE REFLECTION TRANSFORMER */

       RFLXFM = %REFLECTION (VM1, PLN);

    /* APPLY TRANSFORMER TO ALL ATOM LOCATIONS */

       DO I = 1 TO %LASTATOM(MOLQ);
           VM = %GETMVEC(MOLQ,I);
           CALL %SETMVEC(MOLQ,I,VM XFM RFLXFM);
       END;  /* DO I */

    END;  /* REFLECTMOL */
```

G. Implementation of Transformers

This section describes in some detail the implementation of transformers in PL/PROPHET, and may be of interest only to the curious.

Any *similarity* (shape-preserving) *transformation* in 3D space can be achieved by an isotropic scaling about the origin, followed by a rotation about the origin, followed by a translation. Formally:

$$PT = (S * P) * R + T$$

where:

P is the original point (3-vector)
S is a scale factor (FLOAT)
R is a 3x3 rotation matrix
T is a translation (3-vector)
PT is the transformed version of P

In PL/PROPHET, such a transformation is represented by a *transformer*, a 4x3 matrix:

$$\begin{bmatrix} S*R(1,1) & S*R(1,2) & S*R(1,3) \\ S*R(2,1) & S*R(2,2) & S*R(2,3) \\ S*R(3,1) & S*R(3,2) & S*R(3,3) \\ T(1) & T(2) & T(3) \end{bmatrix}$$

or, in formal notation:

$$\left[\begin{array}{c} S * R \\ \hline T \end{array}\right]$$

The XFM operator transforms a point (3-vector) P by multiplying P by the first three rows of the transformer (S*R) and adding the fourth row (T) to the result.

The XFM operator is also used to compute the *composition* of two transformers, thus:

transform1: $PT = (S * P) * R + T$
transform2: $PT' = (S' * P') * R' + T'$

transform1
followed by: $PT' = S'*S*P*R*R' + S'*T*R' + T'$
transform2

In that case, the transformer for the composition is defined (formally) to be:

$$\left[\begin{array}{c} S' * S * R * R' \\ \hline S' * T * R' + T' \end{array}\right]$$

The inverse of a transformation can be computed thus:

$$PT = (S * P) * R + T$$

$$P = 1/S * (PT - T) * R^{-1}$$

$$= 1/S * PT * R^{-1} - 1/S * T * R^{-1}$$

thus the inverse of a transformer (%INVXFM) can be written:

$$\%INVXFM \quad \left(\left[\begin{array}{c} S * R \\ \hline T \end{array}\right] \right) = \left[\begin{array}{c} 1/S * R^{-1} \\ \hline -1/S * T * R^{-1} \end{array}\right]$$

XIII. Miscellaneous

This section consis
system procedures and f
find useful.

A. Input/Output Procedu

1. %GLOBALQUIZ

There are times that
top-level copy of a
without passing it as
%GLOBALQUIZ allows you
sequence is

```
    %GLOBALQUIZ(local-na
            <any dat
             <type>
```

where "local-name" is th
known within the procedu
variable which contains
variable known at top-
supplied in "global-nam
is already a global vari
is illegal), %GLOBALQUIZ
and asks you to supply a

Thus, if you have
declared to be of type G
option of having a top-l
the following statements

```
DECLARE G GRAPH;
DECLARE GNAME TEXT;
```

* * * * * *

Procedures and Functions

s of brief descriptions of nctions that you may sometimes

es and Functions

you will want to obtain a ocal variable of a procedure an argument. The procedure o do this easily. Its calling

```
e, global-name)
>     <text>
```

name by which the variable is e and "global-name" is a TEXT the name by which you want the evel PROPHET. If the name ' is unacceptable (e.g., there ble by that name, or the name informs you about the problem, other name.

a local variable previously APH, and you want to offer an vel copy, you could do it with

```
EAD G;
/* Ask if top-level copy wanted */
IF %YESANSWER('DO YOU WANT A TOP-LEVEL COPY')
   THEN DO;
     /* Ask for Global name */
      GNAME = %GETTEXT('NAME = ',FALSE);
      CALL %GLOBALQUIZ(G,GNAME);
   END;
```

B. Sorting Procedures

In addition to sorting tables using the SORT command discussed in Section IV of the Users Manual, it is possible to sort a table calling the sorting procedure directly. The calling sequence is

```
    %TABLESORT(table, key-specs, sorting-specs)
                      <fixed array> <bool array>
```

where "key-specs" is a FIXED array containing the column numbers of the sort keys, with the major key listed first, and the minor keys (if any) in order of importance. (See Section IV of the Users Manual for a discussion of sorting keys.) "Sorting-specs" is a BOOL array, corresponding to "key-specs". It contains an entry for each sorting key in "key-specs". If an element of the BOOL array is set to TRUE, then the sort for that key will be *ascending*; if the element is set to FALSE, the sort for that key will be *descending*. Only cases of *descending* sorts need to be specified; i.e., if an element of "sorting-specs" is empty, that key will be sorted in ascending order. If the "sorting-specs" argument is entirely empty, then all sorts will be ascending.

Thus to take the example given in Section IV of the Users Manual, to sort the table THERAP with column 2 as the major sort key, columns 3 and 6 the minor keys, and all sorts ascending except for column 6, the command would be

```
    CALL  %TABLESORT(THERAP,[2,3,6],[TRUE,TRUE,FALSE]);
or
    CALL %TABLESORT(THERAP,[2,3,6],[,,FALSE]);
```

Sorting Two-Dimensional Arrays

Two-dimensional arrays containing data of type FLOAT, FIXED, or TEXT can be sorted using the procedure

%ARRAYSORT(array-name,key-specs,sorting-specs)

The last two arguments have precisely the same meaning as in %TABLESORT. Note that a "column" of an array is defined as the *second* index of the array, i.e., when referencing the array A(I,J), *J* is considered to be the column index.

Sorting One-Dimensional Arrays

A one-dimensional array containing data of type FLOAT, FIXED, or TEXT, can be sorted with the procedure

%QSORT(array-name)

All sorts with %QSORT will be ascending.

It is sometimes useful to obtain along with the sorted array an *index* of the sorted permutation. This can be done with the procedure

%QSORTIND(array-name,index-array)

where "index-array" is a FIXED array which after sorting will contain an index of the sorted order. Thus if you had a TEXT array named "ZOO", consisting of

['BAT','DOG','ARMADILLO','CAT']

after calling %QSORTIND(ZOO,INDARA), the index array INDARA would be a FIXED array containing the numbers

[3,1,4,2]

C. Text Functions

1. %TRIM

The function %TRIM removes all carriage-returns, line-feeds, tabs, and form feeds from the argument TEXT and replaces each of them with a space. The "trimmed" TEXT is returned. The calling sequence is:

```
trimmed-text = %TRIM(untrimmed-text)
```

D. Array Procedures

1. %ARRAYPACK

%ARRAYPACK is used to "pack" an array; i.e., to remove any empty elements of the array, leaving only non-empty elements in the array. The original order of the non-empty array elements is preserved.

The calling sequence is

```
CALL %ARRAYPACK(array-name)
```

where "array-name" can be any kind of an array. Thus if X is a FLOAT array containing the numbers [0., ,3.,5., ,0.] with the second and fifth elements empty and you call %ARRAYPACK(X), the array will contain the four numbers [0.,3.,5.,0.].

2. %ARRAYPARE

The procedure %ARRAYPARE can be used to reclaim unused space previously allocated in the array with the ALLOCATE statement (see above, Section V). Using %ARRAYPARE reclaims the space taken up by the empty elements at the <u>top</u> of the array. It does <u>not</u> "pack" the array and empty elements interspersed among non-empty elements are <u>not</u> affected. The calling sequence is

```
CALL %ARRAYPARE(array-name)
```

"Array-name" can be an array of any type.

E. Date and Time Functions

1. Date Functions

It is sometimes useful to be able to convert a representation of a date to numeric form in order to carry out some mathematical operations.

To get a FIXED representation of a date TEXT, you can use the function

```
fixed-date = %DATE2FIXED(date-text)
```

which returns the exact number of days since January 1, in the year 1 A.D. to the specified date.

The TEXT argument passed to %DATE2FIXED must be in the form month-day-year; the subfields may be represented as follows:

MONTH may be:

a) The name of a month, which may be truncated to a minimum of three characters (in fact, only the first three characters are scanned, so MARIGOLD will be interpreted as MARCH) or

b) an integer between 1 and 12.

DAY must be an integer between 1 and the number of days in the month (28, 29, 30, or 31, depending on the month).

YEAR may be:

a) The integer representation of the year, or

b) An integer less than 100, in which case the current century will be added (e.g., 93 is interpreted as 1993), or

c) omitted, in which case the current year is assumed.

The month, day and year subfields may be separated by one or more of the characters space, comma, hyphen, or slash. The following are valid TEXT arguments for use with %DATE2FIXED:

```
'NOVEMBER 25, 1944'
'11/25/44'
'NOVOBIOCIN-25/ 1944'
'NOV 25'          (current year assumed)
```

To convert a FIXED representation of a date into a date TEXT, use the function

```
date-text = %FIXED2DATE(fixed-date)
```

which returns the TEXT value of the date (in the form month-day-year) specified by the FIXED argument "fixed-date".

The following are examples of TEXTs returned by %FIXED2DATE:

```
JAN 31 1976
FEB  4 1976
AUG 12  512
```

Lastly, the day of the week on which a date fell can be determined with the function

```
week-day = %FIXED2WKDAY(date-integer)
```

returns a three character TEXT of the day of the week on which that date fell (or will fall).

For example, the sequence of statements

```
J = %DATE2FIXED('12/7/41');
TYPE  %FIXED2WKDAY(J),%FIXED2DATE(J);
```

will produce the following output on the screen:

```
SUN DEC  7 1941
```

Dates in integer format are amenable to arithmetic manipulation and testing. Two dates may be compared to see which is first chronologically, or subtracted to find the number of days between them. A FIXED number N may be subtracted from or added to an integer date to find the date N days before or after the given date. Moreover, the integer form will result in a substantial economy of storage of dates in tables.

2. Time Functions

Just as the date functions discussed above allow you to convert between TEXT and numeric representation of dates, the time functions described here allow you to perform analogous operations with representations of time.

The function %TIME2SECS converts a TEXT representation of time into a FIXED representation that is the number of seconds since midnight of the argument TEXT. The calling sequence of %TIME2SECS is:

```
seconds = %TIME2SECS(time-text)
```

where "time-text" should be of the form hh:mm:ss [AM/PM]. You can use <u>either</u> a twenty-four hour clock representation, or the suffixes "AM" and "PM". Thus, calling %TIME2SECS with the argument '23:00' will produce the same integer representation as calling it with an argument of '11:00 PM'. Note, too, that minutes and seconds do not have to be specified.

The inverse of the above can be obtained with the function %SECS2TIME which converts the FIXED number of seconds since midnight to a TEXT representation of the time in the format hh:mm:ss. (All times returned by %SECS2TIME will be based on a 24 hour clock.) The calling sequence is

```
time-text = %SECS2TIME(seconds)
```

F. Terminal Functions and Procedures

1. %TERMTYPE

The function %TERMTYPE returns a TEXT which is the type of the terminal you are currently using. Thus, if you are running PROPHET on the standard display terminal, %TERMTYPE will return the TEXT 'TEKTRONIX'. %TERMTYPE has no arguments.

2. %HIGHRES and %NORMRES

The resolution obtained on the display screen is normally sufficient for most PROPHET displays; however, when plotting a PROPHET display, it may sometimes be desirable to get a slightly better screen resolution. The procedure %HIGHRES allows you to do this. (Note that the display on your terminal will <u>not</u> be improved; only the copy sent to the plotter at BBN will be affected.) The calling sequence is simply

```
CALL %HIGHRES
```

You should always reset the screen resolution when you are finished displaying the material to be plotted because using high resolution causes each display to take longer to appear. This is accomplished by calling the procedure %NORMRES.

G. Batch Aids

The functions described here are designed to be of assistance when you are running PROPHET Batch jobs.

1. %OTHERUSER

It is not a good idea to have two PROPHET jobs belonging to the same PPN (project-programmer number) running at the same time. If you have submitted a batch job (see the description of %AUTOBAT in the Public Procedures Notebook), you may want to have that job check to see if another job is logged in under your user number. The function %OTHERUSER can be used to do this. It returns a BOOL which will be set to FALSE if another job is not logged in; and TRUE if you do have another job logged in at that moment.

2. %GETSHIFT

If you do not want your batch job to be run outside of the off-peak hours, the function %GETSHIFT can be used to check on the billing shift at a given moment of the day. %GETSHIFT returns the FIXED number which is the "shift" (for billing purposes) of the day. If you want your batch job to abort if it is no longer running in the off-peak shift, you can use %GETSHIFT to determine what the shift is, and then terminate the batch job. (The simplest way to do this is by having a few question marks typed out.)

Appendix A: Update to the Use

Section III: Basic PL/PROPHET Fun

D. Arithmetic Functions

1. %FACT and %COMBIN func

E. Text Functions

1. %REPLACETEXT function
2. %LEGALNAME function .
3. OCCURS function . . .
4. LOCS function . . . .
5. %REVERSE function . .
6. %CHANGECASE function
7. %COMPSTRING function

G. Input/Output Functions

1. %GETANSWERHELP functi

Section IV: Procedures in PL/PR

A. Arguments of a Procedure

1. %LISTARGS procedure

C. Compiling Procedures

1. %TESTCOMPILED proced

's Guide to PL/PROPHET

ions and Procedures

ions A-1

. . . . A-1

. . . . A-2

. . . . A-2

. . . . A-3

. . . . A-3

. . . . A-3

. . . . A-4

n A-4

PHET

. . . . A-6

re A-6

Appendix A: Update to the User's Guide to PL/PROPHET - 12/84

Section V: Writing and Debugging PL/PROPHET Procedures

C. Common Run-Time Problems

1. %CHECKARGS A-7
2. %EXISTS function A-8

Section VII: Table Functions and Procedures

1. %RACALL and %RACALLQT procedures A-9
2. %FINDCELL and %FINDALLCELLS functions A-9
3. %GETCLIMITS and %SETCLIMITS procedures A-11
4. %WITHINLIMITS procedure A-11
5. %MAKETABFROM procedure A-11
6. %DELETEROWS procedure A-12
7. %TRANSPOSE function. A-13
8. %SETRNUSED procedure and %GETRNUSED function . . A-13
9. %SETCUNIQUE procedure and %GETCUNIQUE function . A-13
10. %SHOWTAB procedure A-14
11. %FORCLAUSE2RANGES procedure A-14
12. %BSEARCHTAB and %BSEARCHROW procedure. A-15
13. %FORMATCOL procedure A-17
14. %SWITCHCOLS procedure. A-17
15. %SWITCHROWS procedure A-17

Appendix A: Update to the User's Guide to PL/PROPHET - 12/84

Section VIII: Display Procedures and Functions

1. %DLINETYPE procedure A-18

2. %POSITIONDISPLAY procedure A-18

Section XII: Vector, Matrix and Transformation Procedures and Functions

1. %MATSIZE procedure A-19

Section XIII: Miscellaneous Procedures and Functions

A. Input/Output Procedures and Functions

1. %RESTORE procedure A-20

2. %GETBOUNDEDFIXED and %GETBOUNDEDFLOAT functions. A-20

3. %SAVE procedure A-21

B. Sorting Procedures

1. %SORTINDEX Procedure A-22

C. Text Functions

1. %TRIMTEXT function A-23

2. %BREAKONCRLF function A-23

3. %WILDMATCH function. A-24

4. %PARAPACK function A-25

5. %TEXT2ARRAY and %ARRAY2TEXT functions A-26

6. %BSEARCHARRAY function A-27

D. Array Procedures

1. $DELETEELEMENTS procedure A-28

2. $INSERTELEMENT procedure A-28

G. Batch Aids

1. $AMIBATCH function A-29

2. $BATCHTIMELEFT function A-29

3. Change to $AUTOBAT A-29

Section III: Basic PL/PROPHET Functions and Procedures

D. Arithmetic Functions

1. %FACT and %COMBIN

The function %FACT returns the factorial of a number (i.e., X!). For example:

```
TYPE %FACT(7)  <GO>
```

The function %COMBIN returns the number of combinations of Y items chosen from a set of X (i.e., X!/Y!(X-Y)!). For example,

```
TYPE %COMBIN(A,B)  <GO>
```

where A and B are FIXEDs which correspond to X and Y in the formula above.

Note that the maximum value of the argument to %FACT is 33. Note, too, that the arguments to both %FACT and %COMBIN should be positive FIXED numbers or positive whole FLOAT numbers. When the argument to %FACT is greater than 9 you should expect to see roundoff errors. Both %FACT and %COMBIN return a FLOAT value.

E. Text Functions

1. %REPLACETEXT

%REPLACETEXT is used to replace a substring within a TEXT variable with a new substring, and return the result in a new TEXT variable. (The original TEXT variable is unchanged.) The general form is:

```
new-txt = %REPLACETEXT(old-val,str1,str2,rep-cnt,start)
```

where the arguments are:

new-txt	the resultant TEXT
old-txt	the original TEXT
str1	TEXT substring to be replaced
str2	TEXT substring to replace "str1"
rep-cnt	FIXED number specifying the number of replacements
start	FIXED indicating where to start searching "old-text" for "str1"

If "rep-cnt" is empty, then all possible replacements of "str1" are made. If "start" is empty, then replacement begins with the first occurrence of "str1". If "str1" is not found in "old-txt", then the text returned is the same as "old-txt".

2. %LEGALNAME

%LEGALNAME is a function that takes a single TEXT argument and returns a BOOL value of TRUE if the TEXT string constitutes a legal variable name in PL/PROPHET and FALSE if it does not. It is called by a statement like:

```
LEGAL = %LEGALNAME(T) <GO>
```

where T is the TEXT variable containing the string being checked, and the function value is assigned to the BOOL variable LEGAL in this case. Legal variable names start with a letter, "%" , or "$" . They do not contain any special characters or blanks, and are not permitted to be one of PROPHET's reserved words. For example, the statement

```
LEGAL = %LEGALNAME('MYTABLE') <GO>
```

sets the BOOL LEGAL to TRUE, while the statement

```
LEGAL = %LEGALNAME('NO-GOOD-NAME') <GO>
```

sets LEGAL to FALSE.

3. OCCURS

OCCURS can be used to find the number of occurrences of one text string within another. For example, to find the number of times "TXT1" occurs within "TXT2" you would type:

```
TYPE OCCURS(TXT1,TXT2)  <GO>
```

There is an optional third argument which may be used to specify where to begin the search. For example, if "START" is of type FIXED, then you could use the following command:

```
TYPE OCCURS(TXT1,TXT2,START)  <GO>
```

For example, the command:

```
TYPE OCCURS('A', 'ATGAACTA', 2)  <GO>
```

returns the fixed number 3.

4. LOCS

LOCS can be used to find the location of each occurrence of one text string in another. It is similar to the function LOC except that rather than returning just the FIRST location of TXT1 in TXT2, LOCS returns a FIXED ARRAY of ALL locations of TXT1 in TXT2. For example, the command:

```
TYPE LOCS('A', 'ATGAACTA') <GO>
```

returns: [1,4,5,8]

There is an optional third argument that can be used to specify where to begin the search. For example, the statement:

```
TYPE LOCS('A','ATGAACTA',2) <GO>
```

returns: [4,5,8]

5. $REVERSE

The function $REVERSE takes a text argument and returns the same text with the characters in reverse order. For example:

```
TYPE $REVERSE('ABCDEF') <GO>
```

returns the text string 'FEDCBA'.

6. $CHANGECASE

$CHANGECASE is a function that takes a text string and converts it to upper or lower case. Note that your terminal must support mixed-case text (the TEKTRONIX 4010 does not). For example, the command:

```
TYPE $CHANGECASE('SMALL','LOWER') <GO>
```

returns the text 'small'. The command:

```
TYPE $CHANGECASE('large', 'UPPER') <GO>
```

returns the text 'LARGE'.

7. %COMPSTRING

%COMPSTRING is a function that takes two text strings as arguments and, while ignoring the upper or lower case of each, compares them for equality. It returns the BOOL TRUE if they are the same and FALSE otherwise.

For example:

```
%COMPSTRING('APPLE', 'apple')  <GO>
```

returns the BOOL TRUE.

The statement:

```
%COMPSTRING('Apple','aPPLe')  <GO>
```

also returns the BOOL TRUE. However,

```
%COMPSTRING('apple', 'appel')  <GO>
```

returns FALSE.

G. Input/Output Functions

1. %GETANSWERHELP

The function %GETANSWERHELP outputs a question on the terminal and waits for the user to respond with one of a pre-selected list of possible answers. It is identical to %GETANSWER except for an additional argument: the help text. The help text is output on the terminal when the user chooses to respond to the initial query with a ?.

The calling sequence is:

```
CALL %GETANSWERHELP(query, ans-array, num-of-answers,
sig-chars, help) <GO>
```

where the arguments are:

query	question to be asked
ans-array	TEXT array of acceptable answers

num-of-answers	FIXED number of acceptable answers
sig-chars	minimum number of characters that will be accepted as a unique match
help	text explanation of options available to the user when the initial response to the query is a ?

Note that if an exact match is desired, then the "sig-chars" argument can be left empty. Also, if you want to allow an EMPTY answer, you should set the "num-of-answers" argument to one more than the actual number of answers contained in "ans-array".

Section IV: Procedures in PL/PROPHET

A: Arguments of a Procedure

1. %LISTARGS

The procedure %LISTARGS is used to obtain the types, in order, of the arguments to the specified procedure. It returns the results in a TEXT ARRAY. The form of the procedure call is:

```
text-array = %LISTARGS(procedure-name) <GO>
```

Here "text-array" is a variable of type TEXT ARRAY, and "procedure-name" is the name of the PROCEDURE, enclosed in single quotes. For example, to see the argument data types for the procedure %GETTEXT, give the command:

```
TYPE %LISTARGS('%GETTEXT') <GO>
```

This statement returns the TEXT ARRAY:

```
['TEXT','BOOL']
```

Note: If there are no arguments to a given procedure, %LISTARGS returns an EMPTY text array.

C. Compiling Procedures

1. %TESTCOMPILED

The function %TESTCOMPILED takes as its argument a procedure and returns TRUE if the procedure is compiled in the current version of PROPHET and FALSE if it is not. For example, if P is a procedure, the calling sequence looks like:

```
bool = %TESTCOMPILED(NOEVAL P) <GO>
```

Note that the expression NOEVAL is required in order not to invoke the procedure.

Section V: Writing and Debugging PL/PROPHET Procedures

C. Common Run-Time Problems

1. %CHECKARGS

%CHECKARGS performs run-time type checking on the arguments of a procedure. It is used as a standard PL/PROPHET statement, and when the procedure is called, it checks that all arguments provided to it are correct. If not, it types a message telling which argument was of an incorrect type, and what the correct argument types should be.

The %CHECKARGS statement should be the FIRST executable statement in the procedure, i.e., it should follow any DECLARE statements. The use of %CHECKARGS can be illustrated by examining the following procedure fragment.

```
MYPROC:  PROCEDURE(A,B,C,D);

DECLARE A FLOAT;
DECLARE B FIXED;
DECLARE C TEXT;
DECLARE D BOOL;
%CHECKARGS(NOEVAL MYPROC);
```

The effect of the %CHECKARGS statement (above) is that when the procedure MYPROC is called, the arguments supplied to it are checked to see if the first is of type FLOAT, the second of type FIXED, etc.

The call to %CHECKARGS is slightly different when it is used to check the arguments to an ENTRY. In this case the statement should be the placed after the ENTRY statement, and takes the form:

```
CALL %CHECKARGS(NOEVAL entry-name,list-of-types)  <GO>
```

Here "entry-name" is simply the name of the entry, and "list-of-types" is the appropriate types of the arguments, separated by commas.

For example, consider that the three arguments to the entry PROCENT below are of type FLOAT, FIXED and TEXT. The call to %CHECKARGS should be written as:

```
PROCENT:  ENTRY(A,B,C);
%CHECKARGS(NOEVAL PROCENT,FLOAT,FIXED,TEXT);
```

2. $EXISTS

$EXISTS is a function that takes a TEXT variable name as an argument and returns the BOOL TRUE if that argument has been declared or used as a top-level variable. $EXISTS will also return TRUE if the reference variable has been declared but is EMPTY. If the referenced variable does not exist at top-level PROPHET, $EXISTS will return FALSE.

Section VII: Table Functions and Procedures

1. %RACALL and %RACALLQT procedures

The procedure %RACALL provides a convenient way to restore all variables and restore and compile all procedures from a specified named-saved-table. Its call takes the following form:

```
CALL %RACALL(table-name,nodef-flag)  <GO>
```

Here "table-name" is the name of the named-saved-table, and "nodef-flag" is a BOOL, which should be TRUE if you want all procedures in the table to be RACed NODEF, and FALSE otherwise.

As each variable in the table is restored, its name is typed on the terminal screen. If a variable of that name already exists in your workarea, you are asked in the usual way if you want it superseded.

For example, to restore all the variables from the table MYSAV, without the definitions of the procedures, the command is:

```
CALL %RACALL(MYSAV,TRUE) <GO>
```

To do the same, but also restoring procedure definitions, the command would be:

```
CALL %RACALL(MYSAV,FALSE) <GO>
```

The procedure %RACALLQT operates in an identical fashion, except that no message is typed on the screen, and any variable currently existing (with the same name as the one being restored) IS superseded. By default, the definitions to the procedures are not restored, as with the nodef-flag above. The form of the procedure call is thus:

```
CALL %RACALLQT(<table-name>)  <GO>
```

2. %FINDCELL and %FINDALLCELLS

The %FINDCELL and %FINDALLCELLS functions are designed to find the location of cells in a table that contain a specified value.

%FINDCELL is used when you want to search a specified row, a specified column, or the entire table to find the FIRST location (if any) of the specified value. The function returns a BOOL which is true if the value is found, and false otherwise.

The calling sequence is:

```
CALL %FINDCELL(tab-name,row-ind,
                    col-ind,val,p-match)  <GO>
```

where the arguments are:

tab-name	table to search
row-ind	row number of table to search or, if no row-ind specified and value is found, row-ind is assigned FIXED row number containing value
col-ind	column number to search or, if no col-ind specified and value is found, col-ind is assigned FIXED column number containing value
val	value being searched for
p-match	BOOL that is TRUE when partial match is acceptable for values of type TEXT and FALSE otherwise

If neither row-ind nor col-ind is specified, then the entire table is searched until the value is found. If both row-ind and col-ind are set, %FINDCELL only checks that single indicated cell.

For example, to find the first (if any) occurrence of the value 2.33 in row 14 of table TAB, the call is:

```
FND = %FINDCELL(TAB,14,IC,2.33,) <GO>
```

where IC is empty. Note that if the value is found (say in column 20) then the BOOL "FND" is true, and IC will contain the value 20. Also note that an empty value for the last argument, "p-match", defaults to FALSE.

The function %FINDALLCELLS is similar to %FINDCELL, however there are two important differences. Firstly, you must specify either a row or a column to search (but not both), and secondly the value returned by the function is a FIXED ARRAY containing the row (or column) numbers of any cell in which the specified value is found. The form of the call is:

```
ind-array = %FINDALLCELLS(tab-name,row-ind,
                          col-ind,val,p-match)  <GO>
```

The arguments are the same as for %FINDCELL, however note that the returned "ind-array" is a FIXED ARRAY (which is empty if no matching cell is found).

3. %GETCLIMITS and %SETCLIMITS

%GETCLIMITS returns a TEXT which is the limits expression of a column of a table. %SETCLIMITS is used to set the limits of a column. The general forms are:

```
CALL %SETCLIMITS(tab,col-id,lim-text)  <GO>

lim-text = %GETCLIMITS(tab,col-id)  <GO>
```

Here "tab" is the table name, "col-id" is either the column number or column heading, and "lim-text" is the TEXT of the limits expression. For example:

```
CALL %SETCLIMITS(MYTAB,2,'X<100.') <GO>
LTXT = %GETCLIMITS(MYTAB,5) <GO>
```

4. %WITHINLIMITS

%WITHINLIMITS is a procedure that is used to determine if a given numeric value satisfies the limits of a column. It returns TRUE if the value falls within the limits, and false otherwise. The general form of the command is:

```
result = %WITHINLIMITS(tab,col-id,value)  <GO>
```

The boolean result will be returned in "result", "tab" is the table in question, "col-id" is the FIXED column number or TEXT column heading, and "val" is the data value to be tested. For example:

```
LIMFLG = %WITHINLIMITS(MYTAB,3,4.67) <GO>
```

Note that %WITHINLIMITS will always return TRUE for columns which do not have limits set, and for any data value which is empty.

5. %MAKETABFROM

%MAKETABFROM can be used within a procedure to accomplish the same effect as a MAKE .. TABLE .. FROM .. FOR expression.

For example:

```
new-tab = %MAKETABFROM(old-tab,row-spec,
col-spec,tab-type,tab-name)  <GO>
```

where the arguments are:

new-tab	the resultant TABLE
old-tab	the source TABLE
row-spec	FIXED ARRAY specifying rows
col-spec	FIXED ARRAY specifying columns
tab-type	TEXT of the table type ('PRIVATE', 'PUBLIC', or 'SHARABLE')
tab-name	TEXT of the new table name

For example,

```
TNEW = %MAKETABFROM(TOLD,RIND,CIND,'PRIVATE','NEWTAB')  <GO>
```

See the description of %FORCLAUSE2RANGES below to see how to generate row-spec and col-spec when using a "for-clause" with a table.

6. %DELETEROWS

%DELETEROWS is a procedure that simultaneously deletes several rows from a table. It is called by a statement like:

```
CALL %DELETEROWS(T,ROWLIST)  <GO>
```

where T is the TABLE, and ROWLIST is a FIXED ARRAY of the rows to be deleted. For example, if you are deleting rows 1,7,19,23 and 8 from a table called MYTABLE, you type:

```
CALL %DELETEROWS(MYTABLE,[1,7,19,23,8])  <GO>
```

Note that when you have several rows to delete from one table, it is a great deal quicker to make one call to %DELETEROWS than it is to make multiple calls to %DELETEROW. (Also, when you use %DELETEROW, you have to worry about the fact that all rows after the row you deleted are renumbered after each call to %DELETEROW.)

7. %TRANSPOSE

The function %TRANSPOSE has been extended to do a transpose for both TABLEs and MATRICES. (A description of the use of %TRANSPOSE with MATRICES is discussed on page 12-8 of the User's Guide to PL/PROPHET.) %TRANSPOSE supersedes the public procedure TRANSTAB.

To use %TRANSPOSE with tables, the command:

```
DISPLAY %TRANSPOSE(table)  <GO>
```

causes the transpose of the argument table to be displayed. Of course, you can use any of the display commands. You can also use %TRANSPOSE directly in an assignment statement:

```
tab2 = %TRANSPOSE(tab1)  <GO>
```

gives the variable "tab2" the value of the transpose of "tab1". The table "tab2" is TEMPORARY.

%TRANSPOSE excludes columns of non-numeric data.

8. %SETRNUSED and %GETRNUSED

The procedure %SETRNUSED enables the rowname capability for a table if the BOOL "useflag" is TRUE, and disables it if "useflag" is FALSE.

The calling sequence is:

```
CALL %SETRNUSED(table,useflag)  <GO>
```

The function %GETRNUSED returns TRUE if the rowname capability of a given table is enabled, and FALSE if it is not.

The calling sequence is:

```
BOOL = %GETRNUSED(table)  <GO>
```

9. %SETCUNIQUE and %GETCUNIQUE

The procedure %SETCUNIQUE enforces uniqueness for the column specified by the FIXED "col-id" if the BOOL "uniqueflag" is TRUE; if "uniqueflag" is FALSE, it disables the feature. Uniqueness may only be applied to columns of type TEXT. If it is enabled, you can not enter duplicate TEXT values in that column.

The calling sequence is:

```
$SETCUNIQUE(table,col-id,uniqueflag)  <GO>
```

The function $GETCUNIQUE returns TRUE if the column specified by the FIXED "col-id" has the uniqueness attribute enabled, otherwise it returns FALSE.

The calling sequence is:

```
BOOL = $GETCUNIQUE(table,col-id)  <GO>
```

10. $SHOWTAB

$SHOWTAB is a procedure that can be used to display a table when you are determining which columns and/or rows of a table you want to have displayed and keep track of the indices in FIXED arrays. (The DISPLAY command does not allow you to specify an array). The calling sequence is:

```
CALL $SHOWTAB(tab,rowspec,colspec,displaytype)  <GO>
```

where the arguments are:

tab	table to be displayed
rowspec	FIXED array of row numbers to be displayed
colspec	FIXED array of col numbers to be displayed
displaytype	TEXT: 'P' for PRINT, 'S' for SCAN, otherwise table will be DISPLAYed

If ROWSPEC is empty, all rows are displayed. Similarly, if COLSPEC is empty, all columns are shown. (See the description of $FORCLAUSE2RANGES to see how to set the rowspec and colspec to the results of applying a for-clause to a table.)

11. $FORCLAUSE2RANGES

$FORCLAUSE2RANGES is a procedure which allows you to apply a "for-clause" to a table at run-time and save the resulting table portions for future use.

The calling sequence is:

```
CALL %FORCLAUSE2RANGES(table, for-clause-text,
     rowspec, colspec, error-flag) <GO>
```

where the arguments are:

table	table name
for-clause-text	TEXT of for-clause
rowspec	FIXED ARRAY of row numbers corresponding to for-clause-text applied to table
colspec	FIXED ARRAY of column numbers corresponding to for-clause-text applied to table
error-flag	BOOL that is set TRUE if for-clause tries to reference a non-existent part of the table or set FALSE when for-clause can properly be applied to table

For example, if at run-time you want to show only those rows of the table "STUDY" where column "SEX" = 'M', you would type:

```
CALL %FORCLAUSE2RANGES
(STUDY,'FOR COLUMN ''SEX'' = ''M''',rowspec,
colspec,error-flag)  <GO>
```

In this example, "rowspec" is set to a FIXED ARRAY of all rows in the table STUDY where column 'SEX' equals 'M'.

The FIXED ARRAYS that result can be used at a later time with other procedures such as %SHOWTAB or %MAKETABFROM. An example is:

```
     %SHOWTAB(STUDY, rowspec, , 'D')  <GO>
```

where rowspec is determined by %FORCLAUSE2RANGES.

12. $BSEARCHTAB and $BSEARCHROW

The procedure $BSEARCHTAB allows you to perform a binary search for a TEXT key within a sorted column of a table. (The advantage of a binary search is that it is much faster than a sequential search.)

NOTE: for a binary search to work,
the column being searched
must be in ascending sorted order!!

The calling sequence for this routine is:

```
index = $BSEARCHTAB(tab,col-id,key)  <GO>
```

where the arguments are:

tab	table containing the data
col-id	FIXED number of the sorted column of type TEXT containing information being searched
key	text for which you are searching, in single quotes

If the key is found, the FIXED index of the row in which it is found is returned; if the key is not located, 0 (zero) is returned.

Note that only exact matches are found, and that if there is more than one occurrence of the key, you are not necessarily given the index of the first.

The procedure $BSEARCHROW has the same calling sequence and arguments as $BSEARCHTAB. It operates in the same way except that rather than returning zero if the key cannot be located, it returns the FIXED row number where the key WOULD BE located if it existed. The "would be" location of a key not found is denoted with a negative sign; the index of a located key is always a positive FIXED number.

13. %FORMATCOL

The procedure %FORMATCOL is used to set the format of a column containing numeric data without having to specify the entire format. It is analogous to the "SET FORMAT" command. The calling sequence is:

```
CALL %FORMATCOL(tab,col-id,dplaces) <GO>
```

where the arguments are:

tab	the table
col-id	the FIXED column number or the TEXT column heading
dplaces	the FIXED number of decimal places or a TEXT of the form 'E4', if an exponential format is desired. If this argument is empty, the resulting format will result in the display of the numbers as integers with no decimal points

(Note that this routine in no way supersedes the system procedure %SETCFORMAT.)

14. %SWITCHCOLS

The procedure %SWITCHCOLS is used to switch the locations of two columns in a table. Its use is similar to the SWITCH COLS command. The form of the procedure call is:

```
CALL %SWITCHCOLS(tab,col-id1,col-id2) <GO>
```

Here "tab" is the name of the table, "col-id1" and "col-id2" are the column numbers or column headings.

15. %SWITCHROWS

The procedure:

```
CALL %SWITCHROWS(tab,row-id1,row-id2) <GO>
```

takes the contents in the ROW designated by the TEXT or FIXED argument "row-id1" and exchanges them with the contents in "row-id2".

Section VIII: Display Procedures and Functions

1. $DLINETYPE

$DLINETYPE is similar to the procedure $DLINE, except that it has an additional argument to specify the kind of line (i.e. dot, dot-dash, etc.) that is to be drawn. The form of the procedure call is:

```
CALL $DLINETYPE(<x1>,<y1>,<x2>,<y2>,<line-type>) <GO>
```

Here "x1", "x2", "y1', and "y2" are the x and y coordinates (of type FLOAT) and "line-type" is a TEXT, having one of the values 'S' (single weight), 'D' (double weight), 'T' (triple weight), 'DOT', 'DASH', 'DOTDASH' OR 'DOTDOTDASH'.

$DLINETYPE will draw a line from (x1, y1) to (x2, y2) in the indicated fashion.

2. $POSITIONDISPLAY

The system procedure $POSITIONDISPLAY is used to arbitrarily position the display of an object on the screen. It is called with a statement like:

```
CALL $POSITIONDISPLAY(OBJECT,XLEFT,YTOP) <GO>
```

where OBJECT is a defined variable of any type that can be displayed (e.g., TEXT, TABLE, MOLECULE, GRAPH, etc.), and XLEFT and YTOP are the screen coordinates of the upper left corner of the display box. Screen coordinates are explained on page 8-1 of the User's Guide to PL/PROPHET. $POSITIONDISPLAY allows you to position the upper left-hand corner of the display above or to the left of the screen, but it aborts with an error message if the upper left corner you specify is to the right of (XLEFT is greater than 511) or below (YTOP is less than -390) the screen.

Section XII: Vector, Matrix and Transformation Procedures and Functions

1. %MATSIZE

%MATSIZE is used to find the dimensions of a MATRIX or two-dimensional array. For example,

```
CALL %MATSIZE(matrix,n-rows,n-cols) <GO>
```

where the arguments are:

matrix	the data object
n-rows	FIXED number of elements in the first dimension
n-cols	FIXED number of elements in the second dimension

Section XIII: Miscellaneous Procedures and Functions

A. Input/Output Procedures and Functions

1. %RESTORE

The procedure %RESTORE is used to restore saved variables from within a procedure that you have written. It is the inverse of %SAVE. The form of the call is:

```
CALL
%RESTORE(<tab-spec>,<var-name>,<val>,<supersede-flag>)  <GO>
```

"tab-spec" can be either the name of a named-saved-table, or one of the TEXTs 'PUBLIC', 'SHARABLE', or 'PRIVATE', indicating that the variable is to be restored from PUBLIC SAVED, SHARABLE SAVED or PRIVATE SAVED respectively. The argument "var-name" is the TEXT name of the variable you want restored, "val" is the locally declared variable of the correct type of the object being restored, and "supersede-flag" is a BOOL which is TRUE if you want to allow the variable to be superseded, and FALSE otherwise.

2. %GETBOUNDEDFIXED and %GETBOUNDEDFLOAT

There are two functions analogous to %GETFIXED and %GETFLOAT which allow a programmer to insist that the number being input by the user must be within a specified range. The calling sequences for these functions are:

```
fixed-num=%GETBOUNDEDFIXED(query,low-val,
                              hi-val,empty-flag)  <GO>
```

and

```
float-num=%GETBOUNDEDFLOAT(query,low-val,
                              hi-val,empty-flag)  <GO>
```

The arguments "query" and "empty-flag" are exactly the same as in the calls to %GETFIXED and %GETFLOAT. The new arguments "low-val" and "hi-val" allow the programmer to specify an inclusive range within which the number being input by the user must fall.

Thus if you are writing a procedure which asks the user to specify the row number of a table for some manipulation, you might use %GETBOUNDEDFIXED in the following way:

```
/* get the number of rows in the table */

NR=%LASTROW(TAB)

/* have the user specify which row he wants */

R = %GETBOUNDEDFIXED('WHAT ROW DO YOU WANT',
                      1,NR,TRUE);
```

In this example, if the user types in a value greater than "NR", the total number of rows in the table, the %GETBOUNDEDFIXED routine objects and requires him to re-enter a number within the range of 1 and NR. %GETBOUNDEDFLOAT works in exactly the same way for FLOAT variables. If low-val is set and hi-val is EMPTY, subsequent responses must be at least as large as low-val. Likewise if lo-val is EMPTY and hi-val is set, the only acceptable responses are those whose values are not larger than hi-val.

3. %SAVE

The procedure %SAVE can be used to do a SAVE command within a procedure. The calling sequence is:

```
CALL %SAVE(tabspec,item,name,replaceflag)  <GO>
```

where the arguments are:

tabspec	either a TEXT specifying which SAVED table to use ('PRIVATE', 'SHARABLE', or 'PUBLIC') or a table itself (if you are using a NAMED SAVED table).
item	the value to be saved (any type)
name	the TEXT name under which you want the variable saved.

replaceflag	BOOL: TRUE if you want it to automatically supersede a saved variable with the same name. FALSE if it should not replace the variable (it will ABORT if a saved variable of the same name is in the saved table and "replaceflag" is FALSE). If "replaceflag" is EMPTY, you will be asked about replacing.

Thus, to save a molecule called M1 in you PRIVATE SAVED table, you might say the following:

```
CALL %SAVE('PRIVATE',M1,'MOL1',TRUE)  <GO>
```

This is equivalent to the top level command:

```
SAVE M1 AS MOL1  <GO>
```

and answering YES if asked if you want to supersede an old variable named MOL1.

B. Sorting Procedures

1. %SORTINDEX

The procedure %SORTINDEX is similar to %TABLESORT and %ARRAYSORT, but does not reset the data in the table or two-dimensional array; rather it returns a FIXED array of the sorted order of the data.

The calling sequence is:

```
new-order=%SORTINDEX(table-or-array,
                          key-specs,sorting specs)  <GO>
```

Note that the arguments are exactly the same as those for %TABLESORT and %ARRAYSORT. The difference is that a call to %SORTINDEX does not reset the data in the table or two-dimensional array, but instead returns a FIXED array which is a permutation (of the row indices) that puts the data in sorted order. Thus, if a table has 5 rows (1,2,3,4,5) and sorting it by the entries in some column would put them in reverse order, the FIXED array returned by %SORTINDEX is [5,4,3,2,1].

C. Text Functions

1. $TRIMTEXT

$TRIMTEXT is a system function that takes a TEXT string and deletes characters as specified from the beginning and end of it. Use a statement of the form:

```
NEWTEXT = $TRIMTEXT(INTEXT,DELLIST)  <GO>
```

where INTEXT is the TEXT value to be trimmed and DELLIST is the TEXT list of characters to trim from either end. The trimmed text is assigned to the TEXT variable called NEWTEXT. For example, the statement:

```
NEWTEXT = $TRIMTEXT(' sample value to trim, ',' ')  <GO>
```

sets NEWTEXT to the value 'sample value to trim,'. The blanks have been trimmed from the beginning and end of the argument. In order to trim both spaces and commas, type:

```
NEWTEXT = $TRIMTEXT(' sample value to trim, ',', ')  <GO>
```

In this example, NEWTEXT is set to the value 'sample value to trim'.

2. $BREAKONCRLF

$BREAKONCRLF is a system function that breaks a TEXT string that spans several lines of text into a TEXT ARRAY with one element for each line in the original TEXT. Use a statement like the following:

```
LINETEXT = $BREAKONCRLF(INTEXT)  <GO>
```

where INTEXT is the TEXT value to be broken up and LINETEXT is the TEXT ARRAY returned by $BREAKONCRLF. Suppose that INTEXT has been assigned a value with the following statement:

```
INTEXT = 'This is the first line
and this is line two.

Line 4 is preceded by a blank line.'  <GO>
```

If INTEXT contains these four lines, then the statement:

```
LINETEXT = $BREAKONCRLF(INTEXT)  <GO>
```

is equivalent to the following series of assignments:

```
DECLARE LINETEXT TEXT ARRAY  <GO>
LINETEXT[1] = 'This is the first line'  <GO>
LINETEXT[2] = 'and this is line two.'  <GO>
LINETEXT[3] = ''  <GO>
LINETEXT[4] = 'Line 4 is preceded by a blank line.'  <GO>
```

In this example, LINETEXT[3] is set to the so-called "null string" to represent the blank line in INTEXT. In order to regenerate the full text string from the TEXT ARRAY, you use the statement:

```
    NEWTEXT = $ARRAY2TEXT(LINETEXT,$CRLF)  <GO>
```

($CRLF is a constant that contains the carriage return-line feed characters that separate each line in a text string.)

NOTE: Calling $BREAKONCRLF is NOT equivalent to calling $TEXT2ARRAY with the second argument set to $CRLF. The statement

```
    LINETEXT = $TEXT2ARRAY(INTEXT,$CRLF)  <GO>
```

creates the TEXT ARRAY LINETEXT with only three elements, since it does not create an element containing the null string for the blank line in INTEXT.

3. $WILDMATCH

$WILDMATCH is a system function that examines each element in a TEXT ARRAY and returns a list of the indices of those elements that contain a specified series of characters in a specified position. For example, in the following statement:

```
    HITLIST = $WILDMATCH(SEARCHSTRING,STRINGLIST)  <GO>
```

SEARCHSTRING is the TEXT search specification as described below, and STRINGLIST is the TEXT ARRAY of values in which to search for this string. HITLIST is set to the FIXED ARRAY that gives the indices of the elements in STRINGLIST that contain the characters specified by SEARCHSTRING.

SEARCHSTRING contains the characters to search for; two special characters, the question mark and the asterisk, provide information on where in each array element to search. If neither of these characters is used, this means to search at the beginning of each element. If an asterisk appears first in SEARCHSTRING, this means to search at the end of each element. Question marks at the beginning or end of SEARCHSTRING indicate the number of characters to count from the beginning or end of each element in doing the search.

Suppose that STRINGLIST has been defined as follows:

```
STRINGLIST =
['FIRST','SECOND','THIRD',
'FOURTH','FIFTH','SIXTH']    <GO>
```

The statement:

```
HITLIST = %WILDMATCH('F',STRINGLIST)  <GO>
```

results in setting HITLIST to the FIXED ARRAY [1,4,5] since elements 1, 4 and 5 of STRINGLIST begin with the character 'F'.

Similarly, the statement:

```
HITLIST = %WILDMATCH('*TH',STRINGLIST)  <GO>
```

sets HITLIST to [4,5,6] since elements 4, 5 and 6 end with 'TH'. Here are some additional examples:

```
%WILDMATCH('?I',STRINGLIST)  <GO>
```

returns the FIXED ARRAY [1,5,6] since these three elements have 'I' as the second character.

```
%WILDMATCH('R??',STRINGLIST)  <GO>
```

returns [1,4] since each of these elements has an 'R' as the third from last character.

```
%WILDMATCH('*O*',STRINGLIST)  <GO>
```

returns [2,4] since 'SECOND' and 'FOURTH' are the two elements that contain the letter 'O'.

4. %PARAPACK

%PARAPACK is a system function that packs a TEXT string so that as many characters as possible are put on each line. Use a statement like the following:

```
NEWTEXT = %PARAPACK(INTEXT,LINEWIDTH,PACKCHAR)  <GO>
```

where INTEXT is the TEXT to be packed, LINEWIDTH is the FIXED number of characters available per line, and PACKCHAR is the TEXT representation of the character that separates the packable pieces in INTEXT (this is usually a blank). In this example, NEWTEXT is set to the TEXT value of the appropriately packed string.

Suppose that the value of INTEXT has been set as follows:

```
INTEXT = 'THIS IS AN EXAMPLE OF A STRING THAT IS TO BE
PACKED ACCORDING TO THE SPECIFICATIONS GIVEN TO $PARAPACK.
     THIS IS A SECOND PARAGRAPH TO ILLUSTRATE THAT ANY
INDENTED LINES RETAIN THE SAME INDENTATION THEY STARTED
WITH.' <GO>
```

The following statement:

```
     TYPE $PARAPACK(INTEXT,50,' ')  <GO>
```

produces the following output:

```
THIS IS AN EXAMPLE OF A STRING THAT IS TO BE
PACKED ACCORDING TO THE SPECIFICATIONS GIVEN TO
$PARAPACK.
     THIS IS A SECOND PARAGRAPH TO ILLUSTRATE THAT
ANY INDENTED LINES RETAIN THE SAME INDENTATION
THEY STARTED WITH.
```

Similarly, the statement:

```
     TYPE $PARAPACK(INTEXT,30,' ')  <GO>
```

produces the following output:

```
THIS IS AN EXAMPLE OF A STRING
THAT IS TO BE PACKED ACCORDING
TO THE SPECIFICATIONS GIVEN TO
$PARAPACK.
     THIS IS A SECOND
PARAGRAPH TO ILLUSTRATE THAT
ANY INDENTED LINES RETAIN THE
SAME INDENTATION THEY STARTED
WITH.
```

5. $TEXT2ARRAY and $ARRAY2TEXT

$TEXT2ARRAY is a function which is used to extract strings of characters from a TEXT variable and store them in the elements of a TEXT ARRAY so that they may be individually accessed. The general form of the function is:

```
     text-array = $TEXT2ARRAY(text-var,delim-text)  <GO>
```

The way in which character strings are to be extracted is determined by the set of delimiter values which are in "delim-text". For example, if "delim-text" consisted of the space character, the comma character and the $CRLF variable (carriage-return, line-feed), then the first element of "text-array" consists of those characters from the beginning of "text-var" up to the first occurrence of either space, comma or $CRLF, the second element whould be the character string between this point and the next occurrence of space, comma or $CRLF, etc. Note that the various characters in "delim-text" are joined together with the concatenation operator.

Some sample $TEXT2ARRAY statements include:

```
TXTVAR = 'AA, BB CC, DEF
         J, K L'   <GO>

DELIM = ','.' '.$CRLF   <GO>

TXTA = $TEXT2ARRAY(TXTVAR,DELIM)   <GO>
```

These statements result in the elements of TXTA as follows:

```
TXTA(1) = 'AA'
TXTA(2) = 'BB'
TXTA(3) = 'CC'
TXTA(4) = 'DEF'
TXTA(5) = 'J'
TXTA(6) = 'K'
TXTA(7) = 'L'
```

If you leave the second argument empty (the delimiter string) then the array elements are the individual characters of the text string, including any delimeters.

$ARRAY2TEXT functions in an inverse fashion. It takes as its arguments a TEXT array and delimiter string, and returns a TEXT. The text is comprised of the individual elements of the array, separated by whatever you specify for the delimiter. For example, consider the TEXT array TXTA above. Then the statement

```
TVAR = $ARRAY2TEXT(TXTA,',')   <GO>
```

gives TVAR the value 'AA,BB,CC,DEI,J,K,L'.

If the delimiter argument is left empty, then the resultant text is a continuous string of characters.

6. $BSEARCHARRAY

$BSEARCHARRAY is used to perform a binary search on an alphabetically sorted text array. $BSEARCHARRAY is called with the TEXT key to search for and the name of the array in which to search. It returns the positive FIXED index of where the key is found in the array. If the key is not found, it returns the negative of the index location where it WOULD be placed if it was a part of the array.

The calling sequence is:

```
INDEX = $BSEARCHARRAY(KEY,ARA)  <GO>
```

where "KEY" is the text key to search for in the sorted TEXT ARRAY "ARA".

Note that only exact matches are found and that if there is more than one occurrence of the key, you are not necessarily given the index of the first.

D. Array Procedures

1. $DELETEELEMENTS

The procedure $DELETEELEMENTS is used to delete a specified list of elements from a one-dimensional array. The proper call is:

```
CALL $DELETEELEMENTS(<array-name>,<list-of-indices>)  <GO>
```

The argument "array-name" is simply the ARRAY variable, and "list-of-indices" is a FIXED ARRAY specifying which elements (i.e., which indices) of the array are to be deleted.

For example, consider the FIXED ARRAY MYARA, which consists of the values: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 and 20. To delete the second, third and tenth elements of the ARRAY MYARA, the command is:

```
CALL $DELETEELEMENTS(MYARA,[2,3,10])   <GO>
```

at which point MYARA consists of the elements: 10, 13, 14, 15, 16, 17, 18, and 20.

2. $INSERTELEMENT

The procedure $INSERTELEMENT is used to insert a specified value into a certain location (i.e. at a certain index) of an ARRAY variable. The proper form of procedure call is:

```
CALL $INSERTELEMENT(<array>,<index>,<value>)  <GO>
```

In the above command, "array" is the one dimensional ARRAY, "index" is the array index where you want the new value inserted, and "value" is the variable to be inserted (of an appropriate type) for the array.

For example, consider the FIXED ARRAY FA which contains the values: 10, 20, 40, 50, 60, 70 and 80.

To insert the value 30 as the 3rd element of FA, the command is:

```
CALL %INSERTELEMENT(FA,3,30)   <GO>
```

at which point FA contains the values: 10, 20, 30, 40, 50, 60, 70 and 80.

G. Batch Aids

1. %AMIBATCH

The %AMIBATCH function returns the boolean value TRUE if it is called from a batch job and FALSE if it is called from a "live" PROPHET job.

2. %BATCHTIMELEFT

The %BATCHTIMELEFT function returns the FIXED number of seconds left before the CPU limit expires. (This limit is set when you submit the Batch job.)

That is, if you set a CPU limit of 20 minutes on your batch job, you can check at various places in the job to make sure that you have sufficient CPU time left to perform the tasks you want done.

3. %AUTOBAT

The minimum amount of runtime for a PROPHET Batch job submitted through the %AUTOBAT command has been changed from 5 minutes to 1 minute.

INDEX

Page

A Format 11-15
%A2C Procedure 7-21
%A2R Procedure 7-22
%A2T Procedure 7-21
Abbreviations, Problems with 5-9
ABORT (see ERROR START) 6-5
ABS Function 3-7
Absolute Value Function 3-7
Account Numbers (see PPNs) 11-7
ACOS Function 3-7
%ADDCOLUMN Procedure 7-15, 7-12
Adding Columns within a Procedure 7-15
Addition of Vectors and Matrices 12-4
ALLOCATE Statement 5-15, 7-17
Allocating Arrays 5-15
Allocating Table Space 7-17
%AMIBATCH Function A-29
AND (Operator) 1-13, 2-8, 5-19
%ANGLE3 Function 12-16
%ANGLE4 Function 12-16
ANY Arrays 1-14
ANY Data Type 1-3, 1-8, 4-8
Arccosine Function 3-7
%ARCHIVECHECK Function 7-23
Arcsine Function 3-7
Arctangent Function 3-7
Arguments, Empty 4-18
Arguments, in Calls 4-18
Arguments, Side Effects on 5-13
Arguments of a Procedure 4-5, A-6
Arguments with Entries 4-11
Arithmetic Errors 5-12
Arithmetic Functions 3-6, A-1
Arithmetic Operators 1-10
Array-Column Transfer 7-21
Array Constants 1-17
Array Functions 3-16
Array Length 1-16
Array Procedures 13-6, A-28
Array-Row Transfer 7-22
Array-Table Transfer 7-19, 7-21
%ARRAY2TEXT Function A-26
%ARRAYLOC Function 3-16
%ARRAYPACK Procedure 13-6
%ARRAYPARE Procedure 13-6
Arrays 1-14, 3-16, 5-15, 7-19, 13-4, 13-6
Arrays, ANY 1-14
Arrays, Assigning Values to 1-15
Arrays, Checking Length of 3-17
Arrays, Declaring 1-16

Page

Arrays, Fixed-Length . . . 1-16
Arrays, Multi-Dimensional . . . 1-15
Arrays, Packing . . . 13-6
Arrays, Paring . . . 13-6
Arrays, Pre-Allocating . . . 5-15
Arrays, Referring to Elements of . . . 1-17
Arrays, Sorting . . . 13-4
Arrays, Square Bracket Notation . . . 1-17
Arrays, Subscripts . . . 1-15
Arrays, Variable Length . . . 1-16
%ARRAYSORT Procedure . . . 13-4
%ARRAYTEST Function . . . 3-16
ASIN Function . . . 3-7
Assigning Values to Arrays . . . 1-15
Assignment Statements . . . 2-1
ATAN Function . . . 3-7
%AUTOBAT Procedure . . . 13-11, A-29

%BASICTYPE Function . . . 3-16
Batch Aids . . . 13-11, A-29
%BATCHTIMELEFT Function . . . A-29
BEGIN . . . END Block . . . 6-3
Block Structure . . . 2-11, 5-3
BOOL Data Type . . . 1-3, 1-6
Boolean Expressions . . . 2-7
Boolean Operators . . . 1-13
Box, Drawing a . . . 8-2
%BREAKONCRLF Function . . . A-23
%BSEARCHARRAY Function . . . A-27
%BSEARCHROW Procedure . . . A-15
%BSEARCHTAB Procedure . . . A-15

%C2A Procedure . . . 7-21, 4-18
CALL Command . . . 4-1, 5-4, 6-1
CAT (Operator) . . . 1-11
CEIL Function . . . 3-7
CFIXED Function . . . 3-4
CFLOAT Function . . . 3-5, 6-4
%CHANGECASE Function . . . A-3
Character Set . . . 1-1
Characters, Control . . . 1-2
Characters, Special . . . 1-1
CHARS Function . . . 3-10
%CHECKARGS Procedure . . . A-7
Checking Data Types . . . 3-12
CLEAR Statement . . . 2-28, 5-4, 5-20
CLOSEIN Statement . . . 11-9, 11-11
CLOSEOUT Statement . . . 11-9, 11-11
Closing Disk Files . . . 11-9, 11-11
Column, Adding a . . . 7-15
Column, Deleting a . . . 7-18
Column, Derivation of a . . . 7-8
Column, Descriptors in a . . . 7-9

Page

Column, Format of a 7-10
Column, Heading of a 7-7
Column, Inserting a 7-15
Column, Type of a 7-7
Column, Unique Heading of a 7-7, 7-15
Column, Width of a 7-10
Column-Array Transfer 7-21
Columns, Number of 7-3
%COMBIN Function A-1
COMMANDLIST Data Type and Subfields 1-19
Comments, Unclosed 5-10
Comments in Procedures 4-1, 5-2
Communicating with Other Computers 11-1, 11-4
Communicating with Other Programs 11-1
COMPILE Command 4-16
Compile-Time Errors 5-7
Compiling Procedures 4-4, 4-16
Complex Data Types 1-18
Compound Statements 2-11
COMPSAVED Procedure 4-17
%COMPSTRING Function A-4
Concatenation of Text 1-11
Conditional Statements 2-7
Constants in PL/PROPHET 1-9
Control Characters 1-2
Conversion of Data Types 3-3, 1-8
COPY Function 3-6, 5-14
COS Function 3-7
COSD Function 3-7
Cosine Functions 3-7
CPU Time Function 3-18
Creating Arrays 1-16
%CRLF (Variable) 3-19
CROSS Operator 12-6, 12-7
CTEXT Function 3-5, 11-12

Data, Formatted 11-12
Data Bases, Organizing 5-1
Data Conversion Functions 3-3
Data types, Checking 3-12
Data Types, Complex 1-18
Data Types, Converting 3-3, 1-8
Data Types, Simple 1-3
Data Types, Subfields 1-18
DATE Function 3-18
Date Functions 13-7
%DATE2FIXED Function 13-7
%DBOX Procedure 8-2
%DCOPY Procedure 8-8
Debugging Procedures 5-22
Declarations, Global 5-5
Declarations, Implicit 4-8
Declarations in Subprocedures 4-14

Page
DECLARE Statement 4-6, 1-9, 1-16, 2-24, 4-2, 5-5, 11-12
Declaring Arrays 1-16
Declaring Variables 4-6, 1-9, 1-16, 5-5
DECODE Function 11-13, 11-18
Definition of a Procedure 4-1
%DELETECOLUMN Procedure 7-18
%DELETEELEMENTS Procedure A-28
%DELETEROW Procedure 7-18
%DELETEROWS Procedure A-12
%DELETETABLE Procedure 7-18
Deleting Columns 7-18
Deleting Rows 7-18
Deleting Tables 7-18
%DELINETYPE Procedure A-18
%DERASE Procedure 8-8
Derived Columns 7-8
Descriptor, Columns of Type 7-9
Designing Procedures 5-1
%DET Function 12-10
Determinant Function 12-10
Dimensions of the Screen 8-1
%DIRECTORYCHECK Function 7-23
Disk Files 11-1, 11-6
Disk Files, Closing 11-9
Disk Files, Input from 11-7
Disk Files, Names of 11-6
Disk Files, Opening 11-8
Disk Files, Output to 11-8
Disk Files, Protection of 11-6
Disk Files, Reading 11-7, 11-10
Disk Files, Writing 11-8, 11-10
Disks 11-1, 11-4
Display Functions and Procedures A-18
Displaying Text 8-3
%DISPLAYSWITCH (Variable) 3-19
DIVISION BY ZERO (Error) 5-12
%DLINE Procedure 8-2
%DMEASURE Procedure 8-6
DO . . . END Block 2-11, 6-3
DO . . . ENDs, Nested 2-12
DO FOREVER Statement 2-21, 6-4, 6-5
DO Loop, Alternate Forms 2-16
DO Loop, Basic 2-13
DO Loop, Mechanics of a 2-13
DO Loop Statement 2-13
DO Loops, Efficient Use of 5-13
DO Loops, Guidelines for 2-17, 5-13, 5-16
DO Loops, Nested 2-15
DO WHILE Statement 2-20
DOEXIT Statement 2-18, 6-3, 6-5
DONEXT Statement 2-19
Dot Notation with Subfields 1-18
DOT Operator 12-6

Page

- %DPOINT Procedure . . . 8-2
  - Drawing a Box . . . 8-2
  - Drawing a Line . . . 8-2
  - Drawing a Point . . . 8-2

- %E (Variable) . . . 3-19
- E Format . . . 11-14, 11-17
- Efficiency Considerations . . . 5-15
- ELSE Statement . . . 2-8, 2-11
- Empty Arguments in Calls . . . 4-18
- EMPTY Function . . . 3-1, 5-11
- Empty Key . . . 3-13
- Empty Variables . . . 3-1, 5-11
- Empty Variables, Testing For . . . 3-1
- ENCODE Function . . . 11-12, 11-16
- ENDs, Unmatched and Extra . . . 5-8
- Entries, Quiet . . . 5-4
- ENTRY Statement . . . 4-11
- %ENTRYTYPE Function . . . 3-17
- ERROR IN INPUT (Error) . . . 2-5
- Error Messages . . . 2-5, 4-18, 5-4, 5-8, 5-11, 5-12, 5-21, 5-23, 6-5
- ERROR START Statement . . . 6-5
- Error Trapping . . . 6-3
- %ERRORFLAG Statement . . . 6-3
- Errors, Arithmetic . . . 5-12
- Errors, Compile-Time . . . 5-7
- Errors, Roundoff . . . 5-12
- Errors, Run-Time . . . 5-11
- EVAL Function . . . 6-1
- %EXISTS Function . . . A-8
- Exiting from DO Loops . . . 2-18
- EXP Function . . . 3-7
- Exponential Function . . . 3-7
- EXT Function . . . 3-10
- Extra ENDs . . . 5-8
- Extracting Text . . . 3-10

- F Format . . . 11-14, 11-17
- %FACT Function . . . A-1
- %FF (Variable) . . . 3-19
- File Names . . . 11-6
- %FINDALLCELLS Function . . . A-9
- %FINDCELL Function . . . A-9
- %FIRSTCOL Function . . . 7-4
- FIXED, Input of . . . 3-13
- FIXED Data Type . . . 1-3
- Fixed-Length Arrays . . . 1-16
- %FIXED2DATE Function . . . 13-8
- %FIXED2WKDAY Function . . . 13-8
- FLOAT, Input of . . . 3-14
- FLOAT Data Type . . . 1-3
- FLOATING OVERFLOW (Error) . . . 5-12

Page

FLOATING UNDERFLOW (Error) . . . 5-12
FLOOR Function . . . 3-7
Footnotes of a Table . . . 7-5
%FORCECOMPILE Command . . . 4-18
%FORCLAUSE2RANGES Procedure . . . A-14
%FORMATCOL Procedure . . . A-17
Formats, Types of . . . 11-14, 11-17, 11-18
Formatted Columns . . . 7-10
Formatted Data . . . 11-12
Formatting Numeric Data . . . 11-8, 11-14, 11-17
Formatting Text . . . 11-15, 11-18
Fortran Code, Converting . . . 2-26
Fortran Programs, Running . . . 11-1
Functions . . . 4-9, 3-1
Functions, Arithmetic . . . 3-6, A-1
Functions, Array . . . 3-16
Functions, Batch Aid . . . 13-11
Functions, Data Conversion . . . 3-3
Functions, Date . . . 3-18, 13-7
Functions, Display . . . 8-1, A-18
Functions, Input/Output . . . 3-13, A-4
Functions, Miscellaneous . . . 13-1
Functions, Terminal . . . 13-10
Functions, Text . . . 3-10, 13-5, A-1, A-23 - 0-27
Functions, Time . . . 3-18, 13-9
Functions, Type-Checking . . . 3-12
Functions, Vector and Matrix . . . 12-9, 12-8

%GETALLFOOTNOTES Function . . . 7-6
%GETANSWER Function . . . 3-14
%GETANSWERHELP Function . . . A-4
%GETBOUNDEDFIXED Functions . . . A-20
%GETBOUNDEDFLOAT Functions . . . A-20
%GETCDERIV Function . . . 7-8
%GETCDESC Function . . . 7-9
%GETCDESCARRAY Function . . . 7-9
%GETCELL Function . . . 7-2, 5-18
%GETCFORMAT Function . . . 7-11
%GETCHEAD Function . . . 7-7
%GETCLIMITS Procedure . . . A-11
%GETCTYPE Function . . . 7-7, 5-18
%GETCUNIQUE Function . . . A-13
%GETCWIDTH Function . . . 7-10
%GETFILE Function . . . 11-7
%GETFIXED Function . . . 3-13
%GETFLOAT Function . . . 3-14
%GETFOOTNOTES Function . . . 7-5
GETLINE Statement . . . 11-10, 11-14
%GETNAME Function . . . 7-3, 6-2
%GETNOTES Function . . . 7-5
%GETRNAME Function . . . 7-4
%GETRNUSED Function . . . A-13
%GETSHIFT Function . . . 13-11

Page

%GETTEXT Function 3-14, 6-3, 13-2
%GETTITLE Function 7-3
%GETTYPE Function 7-3
Global Declarations 5-5
Global Variables 4-8, 5-5, 6-2, 13-1
Global Variables, Guidelines 5-5
%GLOBALQUIZ Procedure 13-1
%GO (Variable) 3-19
GO TO Statement 2-21, 2-23
GRAPH Data Type and Subfields 1-19
%GRAPHCOORDS Function 8-7

Heading of a Column 7-7
Headings, Unique Column 7-7, 7-15
%HIGHRES Procedure 13-10

%IDENTITYMATRIX Function 12-18
IF . . . THEN Statement 2-7
IFs, Nested 2-8, 5-19
ILLEGAL STATEMENT IN BLOCK (Error) 5-8
Implicit Declaration of Variables 4-8
Indenting Code 5-2
INFILE Statement 11-8, 11-11
Initializing Variables 5-11
Input of Disk Files 11-7, 11-10
Input of FIXED 3-13
Input of FLOAT 3-14
Input of Formatted Data 11-13, 11-17
Input of TEXT 3-14
Input/Output Functions 3-13, A-4
INPUT Statement 2-4, 2-25
Input Statements 2-4
INPUT TEXT Statement 2-5
Inputting Variables in Procedures 2-4
%INSERTCOLUMN Procedure 7-15
%INSERTELEMENT Procedure A-28
Inserting Columns 7-15
Inserting Rows 7-16
%INSERTROW Procedure 7-16
Interactive I/O 2-6
Interactive Procedures 5-4
Internal Procedures 4-13, 5-3
Inverse Function 12-8
%INVERSE Function 12-8
%INVXFM Function 12-15

Keys, Sorting 13-3

LABELs 2-23
Largest Numeric Values in PROPHET 3-19, 5-12
LAST LINE PROCESSED Message 5-7
%LASTCOL Function 7-3, 5-18
%LASTROW Function 7-4, 5-18

Page
%%LCLs in Procedures 5-23
%LEGALNAME Function A-2
Length of an Array 1-16, 3-17
Line, Drawing a 8-2
%LISTARGS Procedure A-6
LOC Function 3-11
Local Variables, Index of 5-23
LOCS Function A-3
LOG Function 3-8
LOG10 Function 3-8
Logarithmic Functions 3-8

MAG Function 12-8
Magnitude Function 12-8
%MAGSQD Function 12-8
MAKE PROCEDURE Command 4-4
MAKE TABLE NAMED Command 7-13
%MAKETABFROM Procedure A-111
%MAKETABLE Procedure 7-12, 6-5
%MAKETABLEALLOC Procedure 7-17
Making Tables within Procedures 7-12, 7-17
%MATDIMEN Procedure 12-18
Matrices, Cross Product 12-6, 12-7
Matrices, Dot Product 12-6
Matrices, Miscellaneous Procedures 12-17, 12-16, A-19
Matrix Addition 12-4
MATRIX Data Type 12-2
Matrix Functions 12-9, 12-8
Matrix Multiplication 12-4, 12-6
Matrix Negation 12-4
Matrix Operators 12-5, 12-4
Matrix Subtraction 12-4
%MATSIZE A-19
MAX Function 3-8
%MAXFIXED (Variable) 3-19, 5-12
Maximum Numbers in PROPHET 3-19, 5-12
%MAXNEGFLOAT (Variable) 3-19, 5-12
%MAXPOSFLOAT (Variable) 3-19, 5-12
Measuring Text 8-4
Messages, Error 2-5, 4-18, 5-4, 5-8, 5-11
5-12, 5-21, 5-23, 6-5
MIN Function 3-8
%MINFIXED (Variable) 3-19, 5-12
Minimum Numbers in PROPHET 3-19, 5-12
%MINNEGFLOAT (Variable) 3-19, 5-12
%MINPOSFLOAT (Variable) 3-19, 5-12
Miscellaneous Procedures and Functions 13-1, A-20
Missing Semicolons 5-9
Mnemonic Variable Names 5-3
MOD Function 3-8
Modulo Function 3-8
%MOOLCOORDS Function 8-7
Multi-Dimensional Arrays 1-15

Page
Names, Restrictions on 1-9
Names, Table 7-3
Naming Disk Files 11-6
Naming Procedures 4-4
Naming Variables 1-9, 5-3
Nested DO . . . ENDs 2-12
Nested DO Loops 2-15
Nested IFs 2-8, 5-19
NOCR Output Statement 2-3
NOEVAL Statement 5-6, 5-23, 6-1
%NORMRES Procedure 13-10
NOT (Operator) 1-12
NOTEMPTY Function 3-1, 5-11
Notes of a Procedure 4-16
Notes of a Table 7-5
Numbers, Formatting 11-14, 11-17
Numbers, Largest and Smallest Possible 3-19, 5-12

OCCURS Function A-2
OF Syntax with subfields 1-18
Opening Disk Files 11-8, 11-11
Operators, Arithmetic 1-10
Operators, Boolean 1-13
Operators, Matrix 12-5, 12-4
Operators, Relational 1-12
Operators, Text 1-11
Operators, Vector 12-5, 12-4
Operators in PL/PROPHET 1-10
OR (Operator) 1-13
Organizing Data Bases 5-1
%OTHERUSER Function 13-11
OUTFILE Statement 11-9, 11-11
Output of Formatted Data 11-14
Output Procedures 13-1
Output Statements 2-3
Overflow, Float 5-12

Packing Arrays 13-6
%PARAPACK Function A-25
PARE Command 5-20
Parentheses, Unmatched 5-9
Paring Arrays 13-6
%PERPTO Function 12-8
%PI (Variable) 3-19
PL/PROPHET Statements 2-1, 6-1
Point, Drawing a 8-2
%POSITIONDISPLAY Procedure A-18
PPNs 11-7
Pre-Allocating Arrays 5-15
Pre-allocating Tables 7-17
Precedence of Arithmetic Operators 1-10
Precedence of Boolean Operators 1-14
Precedence of Vector and Matrix Operators 12-7

Page

Procedure, Definition of a 4-1
Procedure, Notes of a 4-16
PROCEDURE Data Type and Subfields 1-19
Procedures 4-1
Procedures, Arguments of 4-5, A-6
Procedures, Array 13-5, A-28
Procedures, Comments in 4-1, 5-2
Procedures, Compilation of 4-16, A-6
Procedures, Debugging 5-22, A-7
Procedures, Declaring Variables in 4-6
Procedures, Designing 5-1
Procedures, Display 8-1, A-18
Procedures, Interactive 5-4
Procedures, Internal 4-13, 5-3
Procedures, Miscellaneous 13-1, A-20
Procedures, Multiple Entries in 4-11, 4-13
Procedures, Naming 4-4
Procedures, Output 13-1
Procedures, Quiet 5-4
Procedures, Re-compiling 4-16
Procedures, Sorting 13-3, A-22
Procedures, Statements in 4-3
Procedures, Table 7-1
Procedures, Terminal 13-10
Procedures, Timing 5-20
Procedures, Using 4-18
Procedures, Writing 5-1, A-7
Procedures, Writing Efficient 5-15
Programming, Structured 2-11, 5-3
Programs, Running Other 11-2
Project-Programmer-Number 11-7
Protection of Disk Files 11-7
Protection of Tables 7-23
%PUTFILE Procedure 11-8
PUTLINE Statement 11-10, 11-12

%QSORT Procedure 13-4
%QSORTIND Procedure 13-4
Quiet Procedures 5-4
Quotes, Unmatched 5-8

%R2A Procedure 7-22
RAC Command 4-4
%RACALL Procedure A-9
%RACALLQT Procedure A-9
RANDOM Function 3-8
Random Number Function 3-8
Random Number Initialization 3-9
%RANINIT Function 3-9
Re-compiling Procedures 4-16
Reading Disk Files 11-7, 11-10
REALTIME Function 3-18
Redundant Code 5-16

Page
$REFLECTION Function 12-15
$REFS Statement 5-23, 6-1
Relational Operators 1-12
$REPLACETEXT Function A-1
Resolution, Screen 13-10
Restore and Compile Command 4-4
$RESTORE Procedure A-20
Restrictions on Variable Names 1-9
RETURN Statement 2-21, 2-27, 4-8, 4-11, 6-4
Returning to PROPHET 11-3
RETURNS Statement 4-6, 4-9, 4-11
$REVERSE Function A-3
$ROTATION Function 12-13
Rotation of Vectors 12-11
Roundoff Errors 5-12
$ROUNDOFF Function 3-9, 5-13
Row, Deleting a 7-18
Row, Inserting a 7-16
Row-Array Transfer 7-22
Rownames, Table 7-4, 7-12
Rows, Number of 7-4
RUN Command 11-2
Run-Time Errors 5-11, A-7
Running Other Programs 11-2
RUNRTN Statement 11-3
RUNTIME Function 3-18, 5-20

SAMPLE Data Type and Subfields 1-19
$SAVE Procedure A-21
$SCALE Function 12-13
$SCALEABOUT Function 12-13
$SCALEV Function 12-13
Scope of Subprocedure Declarations 4-14
Scope of Variables 4-8
Screen Dimensions 8-1
Screen Resolution, Setting 13-10
$SECS2TIME Function 13-9
Semicolons, Missing 5-9
SET Command 2-1, 7-2
SET NOTES of Procedure Command 4-16
$SETALLFOOTNOTES Procedure 7-6
$SETCDERIV Procedure 7-8, 7-15
$SETCDESC Procedure 7-10, 7-15
$SETCDESCARRAY Procedure 7-10
$SETCELL Procedure 7-2, 2-2
$SETCFORMAT Procedure 7-11
$SETCHEAD Procedure 7-7
$SETCLIMITS Procedure A-11
$SETCTYPE Procedure 7-8
$SETCUNIQUE Procedure A-13
$SETCWIDTH Procedure 7-10
$SETFOOTNOTE Procedure 7-5
$SETNOTES Procedure 7-5

Page
%SETRCOL Procedure 7-4
%SETRNUSED Procedure A-13
%SETTITLE Procedure 7-3
Shadow Tables 7-12
Shift, Finding the 13-11
SHORT OF SPACE (Error) 5-4
%SHOWTAB Procedure A-14
Side Effects on Arguments 5-13
%SIGFIG Function 3-9, 5-13
Simple Data Types 1-3
SIN Function 3-9
SIND Function 3-9
Sine Functions 3-9
Size and Space Considerations 5-4
Size of a Table 7-3
Smallest Numeric Values in PROPHET 3-19, 5-12
SORT Command 13-3
%SORTINDEX A-22
Sorting Arrays 13-4
Sorting Procedures 13-3
Sorting Tables 13-3
Space, Pre-allocating Table 7-17
Space, Recovering 2-28
Space/Time Tradeoff 5-15
Special Characters 1-1
SQRT Function 3-9
Square Bracket Notation with Arrays 1-17
Square Root Function 3-9
Statements, Advanced PL/PROPHET 6-1
Statements, Assignment 2-1
Statements, Conditional 2-7
Structured Programming 2-11, 5-3
Subfields of Data Types 1-18
Subprocedures 4-13, 5-3
Subprocedures, Declarations in 4-14
Subscripts, Array 1-15
%SWITCHCOLS Procedure A-17
%SWITCHROWS Procedure A-17
System Variables 3-19

%T2A Procedure 7-19
%TAB (Variable) 3-19
Table, Changing Cells of a 7-2
Table, Deleting a 7-18
Table, Footnotes of a 7-5
Table, Making a 7-12, 7-17
Table, Name of a 7-3
Table, Notes of a 7-5
Table, Paring a 5-21
Table, Pre-allocating Space for a 7-17
Table, Protection of a 7-23
Table, Rownames in a 7-4, 7-12
Table, Size of a 7-3

Page

Table, Sorting a 13-3
Table, Title of a 7-3
Table, Type of a 7-3, 7-12
Table-Array Transfer 7-19
Table Functions and Procedures . . . . 7-1, A-9 - 0-17
Tables, Shadow 7-12
Tables, Temporary 7-12
$TABLESORT Procedure 13-3
Tapes 11-4
Temporary Variables 5-5
Terminal Procedures and Functions 13-10
Terminal Type 13-10
$TERMTYPE Function 13-10
TESTCLOSE Function 11-9
$TESTCOMPILED Procedure A-6
$TESTPROTECT Function 7-23
Text, Concatenating 1-11
Text, Displaying 8-3
Text, Formatting 11-15, 11-18
TEXT, Input of 2-5, 3-14
Text, Measuring 8-4
TEXT Data Type 1-3, 1-6
Text Functions . . . . 3-10, 13-5, A-1, A-23 - 0-27
TEXT Key 2-6
Text Operators 1-11
Text Searching Function 3-11
$TEXT2ARRAY Function A-26
TIME Function 3-18
Time Functions 3-18, 13-9
$TIME2SECS Function 13-9
Timing Procedures 5-20
Title of a Table 7-3
TOO MANY OPEN TABLES (Error) 5-21
TOP Function 3-17
Torsion Angle Function 12-16
$TRACEDEPTH (Variable) 5-22
Transfer, Array-Column 7-21
Transfer, Array-Row 7-21
Transfer, Array-Table 7-19, 7-21
Transfer, Column-Array 7-21
Transfer, Row-Array 7-22
Transfer, Table-Array 7-19
Transformer Operations 12-14
Transformers 12-11
$TRANSLATEV Procedure 12-13
$TRANSLATION Function 12-13
Translation of Vectors 12-11
$TRANSPOSE Function 12-8, A-13
Trapping Errors 6-3
$TRIM Function 13-5
$TRIMTEXT Function A-23
Truth Table 1-13
Type-Checking Functions 3-12

Page

Type-Checking on Input 2-5
TYPE Function 3-12
TYPE NOCR Statement 2-3
Type of a Column 7-7
Type of a Table 7-3, 7-12
TYPE Statement 2-3
Typeout of Variables 2-4
TYPETEST Function 1-8, 3-12

Unclosed Comments 5-10
Undeclared Variables 4-7
UNDEFINED VARIABLE REFERENCE (Error) 5-11, 5-23
Undefined Variables 3-1
Underflow, Float 5-12
%UNIQUECHEAD Function 7-7, 7-15
%UNITVEC Function 12-8
Unmatched ENDs 5-8
Unmatched Parentheses 5-9
Unmatched Quotes 5-8
Using Procedures 4-18

Variables, Declaring 4-6, 1-9, 4-8
Variables, EMPTY 5-11
Variables, Global 4-8, 5-5, 6-2, 13-1
Variables, Implicit Declaration 4-8
Variables, Index of Local 5-23
Variables, Initializing 5-11
Variables, Managing Large 5-4
Variables, Naming 1-9, 5-3
Variables, Scope of 4-8
Variables, System 3-19
Variables, Temporary 5-5
Variables, Undeclared 4-8
Variables in PL/PROPHET 1-9
Vector Addition 12-4
VECTOR Data Type 12-2
Vector Functions 12-9, 12-8
Vector Multiplication 12-4, 12-6
Vector Negation 12-4
Vector Operators 12-5, 12-4
Vector Subtraction 12-4
Vectors, Cross Product 12-6, 12-7
Vectors, Dot Product 12-6
Vectors, Miscellaneous Procedures 12-17, 12-16
Vectors, Rotation 12-11
Vectors, Translation 12-11
Versions of PROPHET 4-16

WHILE Statement 2-20
Width of a Column 7-10
%WILDMATCH Function 7-10
%WITHINLIMITS Procedure A-24
Writing Disk Files 11-8, 11-10

Page
Writing Procedures 5-1
WRONG NUMBER OF ARGUMENTS (Error) 4-18

X Format 11-18
XFM Operator 12-11
XOR (Operator) 1-13

%YESANSWER Function 3-13, 4-2, 13-2

Zero, Division by 5-12

%A2C Procedure 7-21
%A2R Procedure 7-22
%A2T Procedure 7-21
%ADDCOLUMN Procedure 7-15, 7-12
%AMIBATCH Function A-29
%ANGLE3 Function 12-16
%ANGLE4 Function 12-16
%ARCHIVECHECK Function 7-23
%ARRAY2TEXT Function A-26
%ARRAYLOC Function 3-16
%ARRAYPACK Procedure 13-6
%ARRAYPARE Procedure 13-6
%ARRAYSORT Procedure 13-4
%ARRAYTEST Function 3-16
%AUTOBAT Procedure 13-11, A-29
%BASICTYPE Function 3-16
%BATCHTIMELEFT Function A-29
%BREAKONCRLF Function A-23
%BSEARCHARRAY Function A-27
%BSEARCHROW Procedure A-15
%BSEARCHTAB Procedure A-15
%C2A Procedure 7-21, 4-18
%CHANGECASE Function A-3
%CHECKARGS Procedure A-7
%COMBIN Function A-1
%COMPSTRING Function A-4
%CRLF (Variable) 3-19
%DATE2FIXED Function 13-7
%DBOX Procedure 8-2
%DCOPY Procedure 8-8
%DELETECOLUMN Procedure 7-18
%DELETEELEMENTS Procedure A-28
%DELETEROW Procedure 7-18
%DELETEROWS Procedure A-12
%DELETETABLE Procedure 7-18
%DELINETYPE Procedure A-18
%DERASE Procedure 8-8
%DET Function 12-10
%DIRECTORYCHECK Function 7-23
%DISPLAYSWITCH (Variable) 3-19
%DLINE Procedure 8-2
%DMEASURE Procedure 8-6

Page
%DPOINT Procedure 8-2
%E (Variable) 3-19
%ENTRYTYPE Function 3-17
%ERRORFLAG Statement 6-3
%EXISTS Function A-8
%FACT Function A-1
%FF (Variable) 3-19
%FINDALLCELLS Function A-9
%FINDCELL Function A-9
%FIRSTCOL Function 7-4
%FIXED2DATE Function 13-8
%FIXED2WKDAY Function 13-8
%FORCECOMPILE Command 4-18
%FORCLAUSE2RANGES Procedure A-14
%FORMATCOL Procedure A-17
%GETALLFOOTNOTES Function 7-6
%GETANSWER Function 3-14
%GETANSWERHELP Function A-4
%GETBOUNDEDFIXED Functions A-20
%GETBOUNDEDFLOAT Functions A-20
%GETCDERIV Function 7-8
%GETCDESC Function 7-9
%GETCDESCARRAY Function 7-9
%GETCELL Function 7-2, 5-18
%GETCFORMAT Function 7-11
%GETCHEAD Function 7-7
%GETCLIMITS Procedure A-11
%GETCTYPE Function 7-7, 5-18
%GETCUNIQUE Function A-13
%GETCWIDTH Function 7-10
%GETFILE Function 11-7
%GETFIXED Function 3-13
%GETFLOAT Function 3-14
%GETFOOTNOTES Function 7-5
%GETNAME Function 7-3, 6-2
%GETNOTES Function 7-5
%GETRNAME Function 7-4
%GETRNUSED Function A-13
%GETSHIFT Function 13-11
%GETTEXT Function 3-14, 6-3, 13-2
%GETTITLE Function 7-3
%GETTYPE Function 7-3
%GLOBALQUIZ Procedure 13-1
%GO (Variable) 3-19
%GRAPHCOORDS Function 8-7
%HIGHRES Procedure 13-10
%IDENTITYMATRIX Function 12-18
%INSERTCOLUMN Procedure 7-15
%INSERTELEMENT Procedure A-28
%INSERTROW Procedure 7-16
%INVERSE Function 12-8
%INVXFM Function 12-15
%LASTCOL Function 7-3, 5-18

Page

%LASTROW Function 7-4, 5-18
%%LCLs in Procedures 5-23
%LEGALNAME Function A-2
%LISTARGS Procedure A-6
%MAGSQD Function 12-8
%MAKETABFROM Procedure A-111
%MAKETABLE Procedure 7-12, 6-5
%MAKETABLEALLOC Procedure 7-17
%MATDIMEN Procedure 12-18
%MATSIZE A-19
%MAXFIXED (Variable) 3-19, 5-12
%MAXNEGFLOAT (Variable) 3-19, 5-12
%MAXPOSFLOAT (Variable) 3-19, 5-12
%MINFIXED (Variable) 3-19, 5-12
%MINNEGFLOAT (Variable) 3-19, 5-12
%MINPOSFLOAT (Variable) 3-19, 5-12
%MOOLCOORDS Function 8-7
%NORMRES Procedure 13-10
%OTHERUSER Function 13-11
%PARAPACK Function A-25
%PERPTO Function 12-8
%PI (Variable) 3-19
%POSITIONDISPLAY Procedure A-18
%PUTFILE Procedure 11-8
%QSORT Procedure 13-4
%QSORTIND Procedure 13-4
%R2A Procedure 7-22
%RACALL Procedure A-9
%RACALLQT Procedure A-9
%RANINIT Function 3-9
%REFLECTION Function 12-15
%REFS Statement 5-23, 6-1
%REPLACETEXT Function A-1
%RESTORE Procedure A-20
%REVERSE Function A-3
%ROTATION Function 12-13
%ROUNDOFF Function 3-9, 5-13
%SAVE Procedure A-21
%SCALE Function 12-13
%SCALEABOUT Function 12-13
%SCALEV Function 12-13
%SECS2TIME Function 13-9
%SETALLFOOTNOTES Procedure 7-6
%SETCDERIV Procedure 7-8, 7-15
%SETCDESC Procedure 7-10, 7-15
%SETCDESCARRAY Procedure 7-10
%SETCELL Procedure 7-2, 2-2
%SETCFORMAT Procedure 7-11
%SETCHEAD Procedure 7-7
%SETCLIMITS Procedure A-11
%SETCTYPE Procedure 7-8
%SETCUNIQUE Procedure A-13
%SETCWIDTH Procedure 7-10

Page
%SETFOOTNOTE Procedure . . . 7-5
%SETNOTES Procedure . . . 7-5
%SETRCOL Procedure . . . 7-4
%SETRNUSED Procedure . . . A-13
%SETTITLE Procedure . . . 7-3
%SHOWTAB Procedure . . . A-14
%SIGFIG Function . . . 3-9, 5-13
%SORTINDEX . . . A-22
%SWITCHCOLS Procedure . . . A-17
%SWITCHROWS Procedure . . . A-17
%T2A Procedure . . . 7-19
%TAB (Variable) . . . 3-19
%TABLESORT Procedure . . . 13-3
%TERMTYPE Function . . . 13-10
%TESTCOMPILED Procedure . . . A-6
%TESTPROTECT Function . . . 7-23
%TEXT2ARRAY Function . . . A-26
%TIME2SECS Function . . . 13-9
%TRACEDEPTH (Variable) . . . 5-22
%TRANSLATEV Procedure . . . 12-13
%TRANSLATION Function . . . 12-13
%TRANSPOSE Function . . . 12-8, A-13
%TRIM Function . . . 13-5
%TRIMTEXT Function . . . A-23
%UNIQUECHEAD Function . . . 7-7, 7-15
%UNITVEC Function . . . 12-8
%WILDMATCH Function . . . A-24
%WITHINLIMITS Procedure . . . A-11
%YESANSWER Function . . . 3-13, 4-2, 13-2